Technische Universität Chemnitz-Zwickau

DFG-Forschergruppe „SPC“ · Fakultät für Mathematik

G. Haase · T. Hommel · A. Meyer · M. Pester

Bibliotheken zur Entwicklung paralleler Algorithmen

Preprint-Reihe der Chemnitzer DFG-Forschergruppe
„Scientific Parallel Computing“

SPC 94_4 März 1994

zweite, aktualisierte Fassung von SPC 93_1 (Mai 1993), überarbeitet von M. Pester

Inhaltsverzeichnis

BIBLIOTHEKEN ZUR ENTWICKLUNG PARALLELER ALGORITHMEN

Summary. The purpose of this paper is to supply a summary of library subroutines and functions for parallel MIMD computers. The subroutines have been developed at the University of Chemnitz during a period of the last five years. In detail, they are concerned with vector operations, inter-processor communication and simple graphic output to workstations. One of the most valuable features is the machine-independence of the communication subroutines proposed in this paper for a hypercube topology of the parallel processors (excepting a kernel of only two primitive system-dependend operations). They were implemented and tested for different hardware and operating systems including transputer, nCube, PVM. The vector subroutines are optimized by the use of C language and enrolled loops (*BLAS1-like*). The paper includes hints for using the libraries with both Fortran and C programs.

1 Einleitung

In dieser überarbeiteten Fassung wird durch Randmarkierungen auf Änderungen gegenüber der ersten Version hingewiesen.

Die vorliegende Dokumentation verfolgt den Zweck, drei an der TU Chemnitz entwickelte Programmbibliotheken weiteren Interessenten nutzbar zu machen. Hierbei ist der Entwicklungsstandard der einzelnen Teile unterschiedlich.

Die erste Bibliothek `vbasmod` besitzt die längste Nutzungsdauer (seit 1988 etwa) und wurde insbesondere für moderne Blas–like Konzeptionen und für die Sprache C in letzter Zeit verbessert. Ihre gesamten Komponenten haben eigentlich nichts mit der Nutzung auf Parallelrechnern zu tun, sondern benutzen nur die lokale Arithmetik auf einem Prozessor (oder auf jedem Prozessorknoten bei der `SPMD`–Arbeitsweise). Der Hauptnutzen besteht in einer standardisierten Programmgestaltung, bei optimiertem Laufzeitverhalten und notwendiger Benutzung dieser Routinen in der nächsten Hierachiestufe, den Kommunikationsroutinen `Cubecom`. Diese Bibliothek ist eine Sammlung von Programmen, mit denen die typischen Kommunikationsaktionen innerhalb eines `MIMD` Parallelrechners gelöst werden. Gegenüber dem bekannten `PVM` liegt hier ebenfalls eine höhere Hierarchiestufe vor. Wenn man (z.B. bei Workstation–Cluster) die Grundkomponenten der Inter–Prozessor–Kommunikation aus `PVM` besitzt, sind alle Lösungen innerhalb von `Cubecom` direkt ohne Änderung nutzbar durch Anpassung der zwei Basisroutinen `send_chan_0` / `recv_chan_0`. Der konsequente Einsatz der `Cubecom`–Routinen führt zu einer problemlosen Portierbarkeit von parallelen Programmen auf verschiedener Hardware (getestet: Transputer mit verschiedenen Entwicklungsumgebungen, nCube, KSR-1 unter TCGMSG, Paramid i860, Linux, Workstation-Cluster verschiedener Hersteller).

Die dritte Programmsammlung für grafische Ausgaben wurde geschaffen, um mit vergleichsweise einfachen Mitteln passive Grafik, ausgehend von Ergebnissen auf dem Parallelrechner, zu ermöglichen. Diese Bibliothek konnte durch die Umstellung auf die inzwischen auch für Parix verfügbaren Xlib-Funktionen auf dem Parallelrechner eine akzeptable Geschwindigkeit der grafischen Ausgabe bei mittleren Problemgrößen erreichen.

Weiterhin wird zukünftig die Nutzung von spezieller Grafiksoftware auf Workstation angestrebt, so daß die Kopplung völlig anders aufgebaut sein wird.

2 Bibliothek für Vektoroperationen – vbasmod

2.1 Sinn und Zweck der Vektorbibliothek

Die Bibliothek enthält verschiedene, in numerischen Programmen häufig benötigte Operationen mit (langen) Vektoren und wurde zur Verwendung in FORTRAN77 geschrieben, kann jedoch auch aus C heraus benutzt werden (siehe Abschnitt 2.3). Die Rufzeilen der Vektorbibliothek sind allgemeiner gehalten als die Rufzeilen der (oft funktionsidentischen) BLAS1-Routinen [2]. Natürlich könnte man die meisten enthaltenen Vektoroperationen auch direkt im Programmtext als `DO`-Schleifen ausdrücken. Die Nutzung der vorliegenden Bibliothek bietet drei entscheidende Vorteile :

1. Durch die Verwendung von Unterprogramm-, bzw. Funktionsaufrufen wird das rufende Programm übersichtlicher und „lesbarer".

2. Die Realisierung der in der Bibliothek enthaltenen Funktionen und Unterprogramme geht weit über die oben angesprochenen `DO`-Schleifen hinaus, da neben einem *Ausrollen* der enthaltenen Schleifen (*unrolled loops*, BLAS-like) mittlerweile auch die Programmiersprache C und vorhandene BLAS1-Routinen benutzt werden. Durch die damit verbundene Optimierung standardisierter Vektoroperationen sind bei häufiger Verwendung der Bibliotheksroutinen Reduzierungen der Programmausführungszeit auf 50% und weniger keine Seltenheit.

3. Außer dem Bearbeiter der Vektorbibliothek braucht sich kein Anwender der Bibliothek mit lästigen Optimierungen häufig wiederkehrender Programmteile herumzuschlagen.

Die ursprüngliche Version der Bibliothek wurde von Prof. Dr. Arnd Meyer in FORTRAN77 geschrieben. Die BLAS-like Behandlung der Bibliothek erfolgte durch Dr. Matthias Pester. Eine Umsetzung in die Sprache C und die Einbindung von BLAS1-Routinen erfolgte durch Beate Junghans und Gundolf Haase.

2.2 In der Vektorbibliothek enthaltene Routinen

Die in der Vektorbibliothek enthaltenen Routinen werden in vier Gruppen mit identischen Parameterlisten eingeteilt. In eckigen Klammern "`[ ]`" eingeschlossene Zeichenketten sind symbolische Bezeichner, die im entsprechenden Abschnitt näher erläutert werden. Die Größen `N`, `ix`, `iy` und `iz` sind stets vom Typ `INTEGER`, alle anderen Rufzeilenparameter sind der entsprechenden Operation angepaßt.

2.2.1 `V[x][oper](N,X,ix,Y,iy,Z,iz)` - Vektoroperationen - Typ 1

Datentypen : `[x]` = D `REAL*8`
= R `REAL*4`
= I `INTEGER`

Operationen : `[oper]` = `max` Maximum
= `bmax` betragsmäßiges Maximum (vorzeichenrichtig)
= `maxb` Maximum der Beträge
= `min` Minimum
= `plus` Addition
= `minus` Subtraktion
= `mult` Multiplikation
= `div` Division
= `mod` Modulo (nur bei `INTEGER`)
= `0mul` `X:=Y; X(i)=0` bei `Z(i)=0`
= `div0` Division, wobei: `X(i)=0`, falls $|$`Y(i)/Z(i)`$| \leq$ `1E-35`

Die Operationen erfolgen komponentenweise in der Form

`X` := `Y` “`oper`“ `Z` bzw. `X` := “`oper`“(`Y`,`Z`)

mit den Schrittweiten `ix`, `iy`, `iz` und der Komponentenanzahl `N`.
Die verfügbaren Unterprogramme im einzelnen:

```
SUBROUTINE VImax   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRmax   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDmax   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIbmax  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRbmax  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDbmax  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VImaxb  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRmaxb  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDmaxb  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VImin   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRmin   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDmin   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIplus  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRplus  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDplus  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIminus (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRminus (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDminus (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VImult  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRmult  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDmult  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VI0mul  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VR0mul  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VD0mul  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIdiv   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRdiv   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDdiv   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VRdiv0  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VDdiv0  (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VImod   (N,X,ix,Y,iy,Z,iz)
```

2.2.2 `V[x][oper](N,X,ix,Y,iy)` - Vektoroperationen - Typ 2

Datentypen : `[x]` = `I` in Worten (`INTEGER` oder `REAL`)
= `D` Doppelworte (`REAL*8`)
= `R` `REAL*4` (bei `sqrt`)
= `c` Datentypkonvertierung.

Operationen : `[oper]` = `copy` Kopieren
= `chng` Vertauschen
= `sqrt` Quadratwurzel
= `IfromR` Konvertierung `REAL*4` zu `INTEGER`
= `IfromD` Konvertierung `REAL*8` zu `INTEGER`
= `RfromI` Konvertierung `INTEGER` zu `REAL*4`
= `RfromD` Konvertierung `REAL*8` zu `REAL*4`
= `DfromI` Konvertierung `INTEGER` zu `REAL*8`
= `DfromR` Konvertierung `REAL*4` zu `REAL*8`

Die Operationen erfolgen komponentenweise in der Form

`X` := “`oper`“ `Y`

mit den Schrittweiten `ix`, `iy` und der Komponentenanzahl `N`.
Die verfügbaren Unterprogramme im einzelnen:

```
SUBROUTINE VIcopy   (N,X,ix,Y,iy)
SUBROUTINE VDcopy   (N,X,ix,Y,iy)
SUBROUTINE VIchng   (N,X,ix,Y,iy)
SUBROUTINE VDchng   (N,X,ix,Y,iy)
SUBROUTINE VcRfromD (N,X,ix,Y,iy)
SUBROUTINE VcDfromR (N,X,ix,Y,iy)
SUBROUTINE VcIfromD (N,X,ix,Y,iy)
SUBROUTINE VcDfromI (N,X,ix,Y,iy)
SUBROUTINE VcRfromI (N,X,ix,Y,iy)
SUBROUTINE VcIfromR (N,X,ix,Y,iy)
SUBROUTINE VDsqrt   (N,X,ix,Y,iy)
SUBROUTINE VRsqrt   (N,X,ix,Y,iy)
```

2.2.3 `V[x]aXpY(N,X,Y,Alfa,Z)` - Vektoroperationen - Typ 3

Datentypen : [x] = D `REAL*8`
= S `REAL*4`

Es wird komponentenweise, mit der festen Schrittweite 1 die Operation

`X := Y + Alfa * Z`

für alle `N` Komponenten mit der Konstanten `Alfa` durchgeführt.
Konkret stehen die folgenden Unterprogramme zur Verfügung :

```
SUBROUTINE VDaXpY (N,X,Y,Alfa,Z)
SUBROUTINE VSaXpY (N,X,Y,Alfa,Z)
```

Bemerkung : Der Unterschied zu den entsprechenden BLAS-Routinen `DAXPY` und `SAXPY` besteht u. a. darin, daß dort statt `X` und `Y` nur **ein** Parameter vorgesehen ist.

2.2.4 `[x]Scapr(N,X,Y)` - Skalarprodukt

Datentypen : `[x]` = `D` `REAL*8`
= `R` `REAL*4`
= `I` `INTEGER`

Es wird das Skalarprodukt der Vektoren `X` und `Y` (`N` Komponenten) bei fester Schrittweite 1 gebildet.

Konkret stehen die folgenden Unterprogramme zur Verfügung :

```
INTEGER          FUNCTION IScapr (N,X,Y)
REAL             FUNCTION RScapr (N,X,Y)
Double Precision FUNCTION DScapr (N,X,Y)
```

2.3 Wie kann man die Bibliotheken beim Linken einbinden ?

Name der Bibliothek : `libvbasmod.a`
Host : `bezier.mathematik.tu-chemnitz.de`
Pfad : `/usr/global/lib/parix`

Unter Parix kann diese Bibliothek unter Nutzung der Schalter
`-L/usr/global/lib/parix` (Suchpfad für Bibliotheken)
`-lvbasmod` (Stamm des Bibliotheksnamens)
eingebunden werden. Die Linkoption `-L` kann weggelassen werden, falls die Bibliothek wie üblich in `$PARIX/lib` zu finden ist (als symbolisches Link zum Original).

2.4 Portabilität

Aus Gründen der Portabilität existieren drei Quelltextversionen der Bibliothek **vbasmod** mit identischen Rufzeilen. Für den Anwender bleibt dies ohne Bedeutung, da stets die schnellstmögliche Version benutzt und als **libvbasmod.a** vom Verantwortlichen bereitgestellt werden sollte.

Die Quellen (FORTRAN77, C) enthalten keine PARIX- oder transputerspezifischen Teile und können somit auf anderen (Parallel-) Rechnern angewandt werden.

Quellen in	Sprache	nutzbar in	Bemerkung
`~/libs/src/vbasmodf`	FORTRAN	FORTRAN	Grundversion, *unrolled loops*
		C	siehe 2.5.4
`~/libs/src/vbasmodc`	C	FORTRAN	
		C	siehe 2.5.4
`~/libs/src/vbasblas`	C	FORTRAN	BLAS1-Routinen erforderlich
		C	siehe 2.5.4

Die BLAS-Version der Bibliothek ist im wesentlichen mit der C-Version identisch, nutzt aber intern die echten BLAS1-Routinen, wo dies möglich ist.

2.5 Programmbeispiele

2.5.1 Kopieren von Vektoren

Das Verständnis der Wirkungsweise der [`copy`]- und [`chng`]-Routinen wird durch eine Denkweise in 4- bzw. 8-Byte Speichereinheiten erleichtert. Hier wird das Kopieren nur mit `REAL*8`- und `REAL*4`-Vektoren gezeigt, die Operationen mit `INTEGER`-Vektoren sind analog. Da keine arithmetischen Operationen stattfinden, gibt es jeweils nur *eine* Routine fuer `INTEGER` bzw. `REAL*4`.

```
      PARAMETER (lang1=100, lang2=50, lang3=200)
      REAL*8  X(lang3),Y(lang1),Z(2,lang2)
      REAL*4  S(lang1),T(lang1)
                ...
C     Belege Y mit der Konstanten 1 (Konstante => Schrittweite=0)
C
      CALL VDCopy( lang1, Y, 1, 1D0, 0)
C
C     Kopiere auf Z (2*lang2=lang1 !!) Y in umgekehrter Reihenfolge
C
      CALL VDCopy( lang1, Z, 1, Y(lang1), -1)
C
C     Kopiere die 2. Zeile von Z (Schrittweite=Spaltenlaenge) auf
C     jede 3. Komponente von X, mit dem 20. Element von X beginnend.
C
      CALL VDCopy( lang2, X(20), 3, Z(2,1), 2)
C
C     Kopieren des Inhaltes von Y auf S, und dann auf T
C
      CALL VcRfromD( lang1, S, 1, Y, 1)
      CALL VICopy( lang1, T, 1, S, 1)
```

2.5.2 Austausch von Vektoren

```
      PARAMETER (lang=100)
      REAL*4    S(lang),T(lang)
                ...
C     Vertausche Inhalte von S und T
C
      CALL VIchng( lang, S, 1, T, 1)
C
C     Umkehren der Reihenfolge der Feldelemente in S
C
      CALL VIchng( lang/2, S, 1, S(lang), -1 )
```

2.5.3 Vektoroperationen

Zur Illustration von `VDaXpY` und `DScapr` diene das folgende Unterprogramm.

```
C*******************************************************************
C*      SUBROUTINE MultBlQS(n,w,u,sk)
C*
C*      Multipliziert die vollbesetzte, quadratische, symmetrische
C*      Matrix (sk) der Dimension (n) mit einem Vektor (u).
C*      ---> n         : Dimension der Matrix und Vektoren
C*      <--- w         : Ergebnisvektor
C*      ---> u         : Eingangsvektor
C*      ---> sk        : Matrix
C*******************************************************************
C
        SUBROUTINE MultBlQS(n,w,u,sk)

        INTEGER n
        REAL*8  w(*),u(*),sk(*), dscapr

C
C       w = (D+U) * u
C
        ia = 1
        n1 = n
        DO 10 i=1,n
          w(i) = dscapr(n1,u(i),sk(ia))
          ia = ia+n1
          n1 = n1-1
10      CONTINUE
C
C       w = w + L * u
C
        ia = 2
        n1 = n-1
        DO 20 i=2,n
          call VDaXpY(n1,w(i),w(i),u(i-1),sk(ia))
          n1 = n1-1
          ia = ia+n1+2
20      CONTINUE
        RETURN
        END
```

2.5.4 Einbinden der Bibliothek in C-Quellen

Da die Bibliothek mit Blick auf eine Nutzung innerhalb von FORTRAN77-Quellen geschrieben wurde, sind zwei Dinge beim Aufruf aus C-Quellen heraus zu beachten.

1. **Alle** Parameter der Rufzeile müssen als Pointer übergeben werden (Call by Reference).
2. Der Rufname ist in Kleinbuchstaben, ergänzt durch das Unterstreichungszeichen `"_"`, anzugeben. Dies gilt z. B. für die Nutzung unter `PARIX` auf Transputersystemen und für Sun-Workstations, nicht aber für HP-Workstations (ohne Unterstreichungszeichen) oder für `NCUBE` (Großbuchstaben ohne Unterstreichungszeichen).

Zur Illustration diene die C-Version obiger Routine (Abschnitt 2.5.3), die ihrerseits wiederum aus FORTRAN77 heraus mit

```
REAL*8 W(*),U(*),SK(*)
CALL MultBlQS(N,W,U,SK)
```

gerufen werden kann. Die explizite Typangabe für die aus der Bibliothek verwendeten Aufrufe kann durch Einbinden des Headerfiles **vbasmod.h** aus **/usr/global/lib/parix** ersetzt werden.

```
/**********************************************************************/
/**     multblqs_(np,wp,up,skp)                                      */
/**                                                                  */
/**     Multipliziert die vollbesetzte, quadratische, symmetrische   */
/**     Matrix (sk) der Dimension (n) mit einem Vektor (u).          */
/**     ---> n          : Dimension der Matrix und Vektoren          */
/**     <--- w          : Ergebnisvektor                             */
/**     ---> u          : Eingangsvektor                             */
/**     ---> sk         : Matrix                                     */
/**********************************************************************/
void multblqs_(np,wp,up,skp)
 int    *np;
 double *wp,*up,*skp;

{unsigned long  n= *np, i,n1;
 double *w=wp, *u=up, *sk=skp;
 double dscapr_();
 void   vdaxpy_();
/*                                        w = (D+U) * u            */
 for (n1=n, i=0; i<n; sk += n1-- , i++)
     *(w+i) = dscapr_(&n1,u+i,sk);
/*                                        w = w + L * u            */
 for (n1=n-1, i=1, u=up-1, w=wp, sk=skp+1; i<n; sk += --n1 +2 , i++)
      vdaxpy_(&n1,w+i,w+i,u+i,sk);
}
```

3 Kommunikationsbibliothek – Cubecom

3.1 Grundlagen

Wir beschäftigen uns mit MIMD-Parallelrechnern, einer Anzahl von Prozessoren mit lokalen Speichern, die über ein Kommunikationsnetzwerk Daten miteinander austauschen können.

Dieses Kommunikationsnetzwerk ist in unserem Fall ein Hypercube der Dimension `NCUBE`. Jeder der `NPROC` $= 2^{\texttt{NCUBE}}$ Prozessoren hat genau `NCUBE` „direkte“ Nachbarn.

Die Kommunikation zwischen direkten Nachbarn wird auf bekannten Parallelrechnern durch spezielle Systemroutinen unterstützt, z. B.:

Transputer 3L :	`F77_Chan_Out_Message(...)`	`F77_Chan_In_Message(...)`
Transputer Helios :	`PSX_Write (...)`	`PSX_Read (...)`
Transputer Parix :	`Send (...)`	`Recv (...)`
nCube :	`nWrite (...)`	`nRead (...)`
WS-Cluster (PVM):	`pvmfsend (...)`	`pvmfrecv (...)`

Die Beispiele zeigen, wie zweckmäßig es aus Anwendersicht ist, eine weitgehend hardware- bzw. system-unabhängige Kommunikationsbibliothek zu schaffen. Zu diesem Zweck wurden die folgenden beiden Routinen zur elementaren Kommunikation zwischen direkten Nachbarn eines Hypercubes als Grundbausteine der Kommunikationsbibliothek verwendet, die jeweils an die aktuelle Hardware bzw. Systemsoftware angepaßt wurden:

```
SUBROUTINE SEND_CHAN_0 ( nWords, Data, LinkNo )
SUBROUTINE RECV_CHAN_0 ( nWords, Data, LinkNo )
```

Dabei ist `LinkNo` die logische Link-Nummer des Prozessors im Hypercube, welche die Verbindung zum gewünschten Nachbarn realisiert. Die Abbildung dieser logischen Nummer auf physische Links des Prozessors ist nicht Aufgabe des Anwenders. Alle weiteren Kommunikationsroutinen basieren auf diesen beiden und sind somit leicht auf andere Parallelrechner zu portieren. Wir gehen dabei stets von einer *synchronen* Kommunikation aus, auch wenn auf einzelnen Parallelrechnern beispielsweise die `SEND`-Operation gepuffert ablaufen kann.

Bei den genannten Routinen muß die Länge des zu sendenden bzw. zu empfangenden Datenfeldes auf beiden beteiligten Prozessoren vorher bekannt sein. Soll dem Zielprozessor die (*variable*) Länge des Datenpakets erst mitgeteilt werden, so ist diese als extra Datenpaket (1 Wort) zu senden. Diese Funktionalität wird durch die folgenden (jetzt bereits maschinenunabhängigen) Routinen realisiert.

```
SUBROUTINE SEND_CHAN ( nWords, Data, LinkNo )
SUBROUTINE RECV_CHAN ( nWords, Data, LinkNo )
```

Die Länge `nWords` wird als Information mit gesendet und ist somit für `RECV_CHAN` ein Output-Parameter; sie darf auch 0 oder negativ sein – dann wird **nur** diese Information übermittelt.

Für spezielle Anwendungen, deren Kommunikationsbedarf (*teilweise*) von der Hypercube-Topologie abweicht, wurden die folgenden Routinen in die Bibliothek aufgenommen:

```
SUBROUTINE SEND_NODE_1 ( nWords, Data, NrTo )
SUBROUTINE RECV_NODE_1 ( nWords, Data, NrFrom )
```

Hier wird die *Prozessornummer* (gemäß Hypercube-Numerierung) anstelle der *Linknummer* zur Bestimmung des Kommunikationspartners verwendet. Es ist nicht erforderlich, daß beide Prozessoren direkt benachbart sind. Eine solche Fernkommunikation wird heute von den meisten Parallelrechnersystemen unterstützt.

Alle im weiteren aufgeführten Routinen verwenden jedoch die für den Hypercube typischen link-orientierten Kommunikationsroutinen.

Für den oft auftretenden Datenaustausch zwischen zwei benachbarten Prozessoren (je ein `SEND` und `RECV`) kann z. B. die folgende Routine verwendet werden:

```
SUBROUTINE EXCHNG ( LinkNo, nWords, SendData, RecvData )
```

Diese Routine gewährleistet einen Deadlock-freien Datenaustausch, ohne das rufende Programm mit diesem Problem zu belasten.

Ein weiterer systemabhängiger Teil besteht in der Initialisierungsphase des parallelen Programms. Hier sind u. a. notwendige Vorbereitungen für die Nachbarschaftskommunikation zu treffen; die einzelnen Programme auf den Prozessoren müssen sich selbst identifizieren, d. h. sie ermitteln die eigene Prozessornummer und die Gesamtzahl der benutzten Prozessoren, bzw. die aktuelle Dimension des Hypercubes. Diese Aufgabe realisiert das folgende, ebenfalls rechner- und systemabhängig zu modifizierende Unterprogramm

```
SUBROUTINE TRINIT ( iHelp )
```

wobei der `Integer`-Parameter `iHelp` lediglich zu Testzwecken unter Parix verwendet wird: Im Falle `iHelp=1` erfolgt eine 1:1-Zuordnung der Numerierung der Prozessoren durch Parix zu der des Hypercubes, anderenfalls eine Zuordnung mit einer bestmöglichen Ausnutzung der physisch geschalteten Links des Parix-Netzes im Hypercube.

Für den Fall, daß ein Parallelrechner eine spezielle Endebehandlung erfordert (z. B. PVM), wurde ein zusätzliches Unterprogramm `TRCLOSE` zum Beenden des Hauptprogramms (auf jedem Prozessor) vorgesehen.

Für die Arbeit mit der Bibliothek sollte der Nutzer folgende **Variablenkonvention** berücksichtigen:

> Die „topologischen“ Parameter des aktuellen Hypercubes stehen dem Anwenderprogramm über `COMMON`-Blöcke zur Verfügung, die mittels
>
> `INCLUDE 'trnet.inc'`
>
> in den Quelltext eingebunden werden können. Damit kann man (**nach** dem Aufruf von `TRINIT`) auf diese Werte über die folgenden Variablen zugreifen:
>
> | | | |
> |---|---|---|
> | `NCUBE` | – | Hypercube-Dimension (Anzahl der Links je Prozessor) |
> | `NPROC` | – | Anzahl der Prozessoren (natürlich ist `NPROC`= $2^{\texttt{NCUBE}}$) |
> | `ICH` | – | eigene Prozessornummer (0,...,`NPROC`−1)
Prozessor 0 ist derjenige, der mit dem Host-Rechner direkt kommunizieren kann (bzw. sollte) |
> | `ICHRING` | – | Position des Prozessors innerhalb des in den Hypercube eingebetteten Prozessorrings. (Für `ICH` = 0 ⇒ `ICHRING` = 0) |
> | `LFORW` | – | Linknummer nach „vorn“ im Prozessorring |
> | `LBACK` | – | Linknummer nach „hinten“ im Prozessorring |
>
> Die Prozessoren werden mit Null beginnend numeriert, die Links auf dem einzelnen Prozessor sind von `1` bis `NCUBE` numeriert und jeweils mit einem Link

gleicher Nummer auf dem Nachbarprozessor verbunden (vgl. übliche Hypercube-Topologie, Links `0` bis `NCUBE-1`).

Auf dieser Grundlage beruhen die für die Realisierung paralleler Algorithmen eigentlich interessierenden Kommunikationsroutinen für *globale* Operationen im Hypercube:

`TREE_DOWN`	schnelles *Broadcasting* der Daten von Prozessor 0 zu allen anderen Prozessoren
`TREE_UP`	schnelles *Zusammenfassen* der Daten von allen Prozessoren zum Prozessor 0 (Speicherkapazität beachten!)
`CUBE_CAT`	*Einsammeln* von Daten variabler Länge von allen Prozessoren zu allen Prozessoren
`CUBE_EXCH`	nach Prozessornummern geordnetes *Einsammeln* von Daten gleicher Länge von allen Prozessoren zu allen Prozessoren
`CUBE_DOD`	Ausführung einer Operation *über alle Prozessoren* für `DOUBLE PRECISION`-Operanden
`CUBE_DOI`	Ausführung einer Operation *über alle Prozessoren* für `REAL`- oder `INTEGER`-Operanden
`TREEUP_DOD`	Ausführung einer Operation *über alle Prozessoren* für `DOUBLE PRECISION`-Operanden; Ergebnis entsteht nur auf Prozessor 0
`TREEUP_DOI`	Ausführung einer Operation *über alle Prozessoren* für `REAL`- oder `INTEGER`-Operanden; Ergebnis entsteht nur auf Prozessor 0
`CUBE_DIM`	Festlegung einer neuen Hypercube-Dimension während des Programmlaufs, im Rahmen der beim Start angeforderten (maximalen) Prozessoranzahl
`RING_OUT`	sequentielles *Durchreichen* der Daten von allen Prozessoren zum Prozessor 0 (im Zusammenspiel mit der Routine `RING_RECEIVE0`)
`RING_FORW`	Bezüglich des eingebetteten Prozessor-Rings sendet jeder Prozessor Daten an den Nachfolger und empfängt Daten von seinem Vorgänger
`RING_BACK`	Bezüglich des eingebetteten Prozessor-Rings sendet jeder Prozessor Daten an den Vorgänger und empfängt Daten von seinem Nachfolger

Bei allen genannten Operationen wird vorausgesetzt, daß sie „**zugleich**“ auf allen Prozessoren gerufen werden (mit einer kleinen Ausnahme bei `RING_OUT`). Dabei bezieht sich „**zugleich**“ auf den **logischen Ablauf** des Programms, **nicht** auf den exakten Zeitpunkt (die Prozessoren warten zwecks Kommunikation bei Bedarf freiwillig aufeinander).

Darüber hinaus stehen einige Routinen zur Zeitmessung als Utilities für die Testphase von Programmen zur Verfügung, die im Anschluß an die Beschreibung der Kommunikationsroutinen kurz vorgestellt werden.

Am Ende dieses Abschnitts folgt eine kurze beispielhafte Erläuterung zur Wirkungsweise der Kommunikationsroutinen.

3.2 Beschreibung der Kommunikationsroutinen

3.2.1 `TREE_DOWN` - Senden vom Prozessor 0 an alle anderen Prozessoren

Rufzeile: `call TREE_DOWN ( N, X )`

Parameter:

- `N` - Länge des zu sendenden (bzw. empfangenen) Datenfeldes; gezählt in Worten
- `X` - Vektor mit `N` Worten (bzw. $\frac{N}{2}$ Doppelworten), der von Prozessor 0 gesendet und von allen anderen Prozessoren empfangen wird

Funktion: Daten, die Prozessor 0 im Dialog mit dem Nutzer am Hostrechner (oder aus Eingabefiles) erhalten hat, müssen allen Prozessoren zugänglich gemacht werden. Dies geschieht hier durch ein baumartiges Versenden im Hypercube (in `NCUBE` Zeitschritten). **Zu beachten** ist, daß beide Parameter für alle Prozessoren `ICH` > 0 Output-Parameter sind, d. h. insbesondere, daß `N` nie als Konstante angegeben werden sollte.

Tip: Mit `N=0` dient das Programm einer gewissen Synchronisation (z. B. vor einer Zeitmessung anzuwenden); dabei ist gesichert, daß alle Prozessoren ihre Arbeit erst fortsetzen, wenn Prozessor 0 dazu das *Signal* gibt.

Beispiel: Typische Programmierung eines Dialogs im Hauptprogramm mit anschließender Übergabe der Daten an alle Prozessoren:

```
         ...
        PARAMETER   ( L_INFO = 2, L_DATA = 100 000 )
        DIMENSION   INFO (L_INFO), HELP (L_DATA)
        DOUBLE PRECISION          HELP
        EQUIVALENCE ( INFO(1), N ), ( INFO(2), EPS )
        CHARACTER*20 FNAME
        include 'trnet.inc'

        call TRINIT ( Help )
        IF ( ICH .EQ. 0 ) THEN
           write (*,'(A,$)')    ' Dimension, Epsilon = '
           read (*,*) N, EPS
           IF ( N .GT. 0 ) THEN
             write (*,'(A,$)') ' Eingabedatei: '
             read (*,'(A)') FNAME
             OPEN  (UNIT=10, FILE=FNAME, FORM=UNFORMATTED)
             READ (10) (HELP(I),I=1,N)
             CLOSE (UNIT=10)
           END IF
           L = L_INFO
        END IF
        call TREE_DOWN (L,INFO)
  C Variablenwerte N und EPS nun auf allen Prozessoren bekannt
        IF ( N .GT. 0 ) THEN
          L=2*N
          call TREE_DOWN (L,HELP)
           ...
        ELSE
          call TRCLOSE
        ENDIF
```

3.2.2 `TREE_UP` - Zusammenfassung von Daten aller Prozessoren

Rufzeile: `call TREE_UP ( Nlocal, Xin, Nout, Xout, MaxX )`

Parameter:

`Nlocal` - lokale Länge von `Xin` (in Worten),
`Xin` - Datenfeld, das vom aktuellen Prozessor geliefert wird
`Nout` - Gesamtlänge des Datenfeldes nach der Zusammenfassung (nur auf Prozessor 0 richtig belegt)
`Xout` - „hinreichend“ großes Feld, auf dem die Daten von allen Prozessoren hintereinander gespeichert werden können
`MaxX` - maximal zur Verfügung stehende Länge des Feldes `Xout`

Funktion: Daten, die zunächst verteilt auf allen Prozessoren vorliegen, sollen aneinandergereiht zum Prozessor 0 gesendet werden. Dies erfolgt durch ein baumartiges Zusammenfassen der Daten von jeweils zwei Prozessoren in `NCUBE` Kommunikationsschritten mit zunehmender Länge des Vektors `Xout`. Das Ergebnis auf Prozessor 0 ist die nach der Prozessornummer `ICH` geordnete Aneinanderreihung der lokalen Datenpakete.

Diese Routine ist offensichtlich nur für kleine Datenpakete geeignet, da sonst bei höheren Prozessoranzahlen enorme Speicherplatzprobleme entstehen.

Die Größe `MaxX` wird vorläufig nicht berücksichtigt, da im Falle eines nicht ausreichenden Arbeitsfeldes `Xout` eine komplizierte Fehlerbehandlung notwendig wird.

Beispiel: Jeder Prozessor hat ein Teilgebiet zu diskretisieren und besitzt lokal eine Anzahl `NI` von inneren Knoten und eine Anzahl `NC` von Koppelknoten. Diese Informationen soll Prozessor 0 ausgeben.

```
        ...
      PARAMETER ( MAX_PROC = 512 )
      INTEGER NCNI (2,MAX_PROC), NH(2)
      include 'trnet.inc'
        ...
      NH(1)=NC
      NH(2)=NI
      call TREE_UP ( 2,NH,Nout,NCNI,2*MAX_PROC )
      IF ( ICH .EQ. 0 ) THEN
        write (*,*)    ' Knotenanzahl auf allen Prozessoren:'
        write (*,100) (I-1,NCNI(1,I),NCNI(2,I), I=1,NPROC)
100     Format ( ' Prozessor', I3, ':', 2I8 )
      ENDIF
        ...
```

3.2.3 `CUBE_CAT` - globaler Austausch variabler Datenfelder

Rufzeile: `call CUBE_CAT ( Nlocal, Xin, Nout, Xout)`

Parameter:

`Nlocal` - Länge des Feldes `Xin`, das der Prozessor liefert; gezählt in Worten (!)
`Xin` - Datenfeld, das der aktuelle Prozessor allen anderen zu schicken hat
`Nout` - Länge des Output-Vektors (Summe aller `Nlocal`)
`Xout` - Feld, auf dem die Daten von allen Prozessoren gesammelt werden können (hinreichend lang)

Da die Daten nur zu kopieren sind, spielt der Typ von `Xin` keine Rolle, lediglich bei `DOUBLE PRECISION` ist zu beachten, daß `Nlocal` die doppelte Zahl der Elemente angeben muß.

Funktion: Die lokalen Daten von jedem Prozessor werden an alle Prozessoren verteilt und zu einem Vektor `Xout` zusammengefaßt. Dabei stehen die Daten, die der eigene Prozessor beizutragen hat, am Anfang des Vektors. Die Vektorkomponenten sind somit auf jedem Prozessor in unterschiedlicher Reihenfolge angeordnet. Falls eine genaue Zuordnung notwendig ist, muß diese z. B. über Indexvektoren geschehen, die in gleicher Weise zusammengesetzt werden. Der Vorteil dieser Routine liegt vor allem in der Möglichkeit, auf jedem Prozessor unterschiedlich lange Datenfelder bereitzustellen (auch Länge `Nlocal=0`).

Beispiel: Austausch von Daten für Koppelknoten bei FEM

```
       ...
      INTEGER Iglob(*), Ihelp(*)
      DOUBLE PRECISION  RC(*), HELP(*)
       ...
C globale Nummern der Koppelknoten der einzelnen Prozessoren
C austauschen :
      CALL Cube_Cat ( NC, Iglob, Nout, Ihelp)
       ...
C Laenge in Worten, deshalb  2*NC  bzw. Nakt/2 :
      CALL Cube_Cat ( 2*NC, RC, Nakt, Help )
      Nakt=Nakt/2
C mit VECADI die lokal interessierenden zu akkumulierenden Daten
C entsprechend der Indizes Ihelp(*) aus dem (globalen) Vektor HELP
C heraussuchen :
      CALL VDcopy ( NC, RC, 1, 0D0, 0)
      CALL VECADI ( NC, RC, Nakt, Help, Ihelp)
       ...
```

3.2.4 `CUBE_EXCH` - Austausch von Datenpaketen gleicher Länge

Rufzeile: `call CUBE_EXCH ( Nlocal, Xin, Xout, ID )`

Parameter:

`Nlocal`	-	Länge des Feldes `Xin`, das der Prozessor liefert; gezählt in Worten (!)
`Xin`	-	Datenfeld, das der aktuelle Prozessor an alle anderen zu schicken hat
`Xout`	-	Feld, auf dem die Daten von allen Prozessoren ankommen, Länge ist `NPROC` $*$ `Nlocal`
`ID`	-	eigene Prozessornummer; darf entweder `ICH` oder `ICHRING` sein (vgl. Abschnitt 3.1)

Funktion: Die verteilt gespeicherten Teile eines globalen Vektors werden entsprechend der angegebenen Prozessornummer `ID` geordnet von allen Prozessoren zusammengefaßt und liegen anschließend auf jedem Prozessor (in identischer Reihenfolge der Komponenten vor). Die Ordnung nach Prozessornummer funktioniert nur für die beiden möglichen Werte `ICH` (Hypercube-Numerierung) oder `ICHRING` (Numerierung im eingebetteten Ring).

Diese Routine kann nur dann verwendet werden, wenn `Nlocal` auf allen Prozessoren gleich ist.

Beispiel: Zeitaufwendige Berechnung eines Vektors, den alle Prozessoren benötigen, lokal ausführen:

```
        ...
      include 'trnet.inc'         ! Hypercube-Parameter
      DOUBLE PRECISION X(*)       ! Laenge:  NPROC*Nlocal
        ...
C Berechnung der Komponenten X(1), ..., X(Nlocal)
C mit X(i) = f ( ICH*Nlocal + i )
        ...
      call CUBE_EXCH ( Nlocal, X, X, ICH )
C Der Vektor X steht nun in voller Laenge auf allen Prozessoren
C mit X(i) = f ( i )
        ...
```

3.2.5 `CUBE_DOD/CUBE_DOI` - Operationen über alle Prozessoren

Rufzeile: `call CUBE_DOx ( N, X, Y, H, VtOP )`

Parameter:

`N` - Länge der Operandenfelder je Prozessor (oft ist `N=1`)
`X` - Resultatvektor (Ergebnis der Operation über alle Prozessoren)
`Y` - Operanden-Vektor des aktuellen Prozessors
`H` - Hilfsvektor, der zur Prozessorkommunikation erforderlich ist
`VtOP` - Name einer Vektoroperation (siehe Abschnitt 2.2.1) mit der Rufzeile: `call VtOP (N,X,ix,Y,iy,Z,iz)`

Die Vektoren sind vom Typ `DOUBLE PRECISION` für `x = D` bzw. `REAL` oder `INTEGER` für `x = I`

Funktion: Jeder Prozessor hat lokal Daten berechnet (Vektor `Y`), die nun über alle Prozessoren mittels einer (kommutativen) Operation `VtOP` verknüpft werden. Dabei steht „`t`“ für `D` oder `R` oder `I`, entsprechend dem Typ der Operanden, und „`OP`“ für eine der (komponentenweise auszuführenden) Operationen:

`PLUS` - Summe über alle Prozessoren
`MULT` - Produkt über alle Prozessoren
`MAX` - Maximum von allen Prozessoren
`MIN` - Minimum von allen Prozessoren
`MAXB` - Maximum der Absolutbeträge
`MINB` - Minimum der Absolutbeträge
`BMAX` - vorzeichenrichtiger betragsgrößter Wert
`BMIN` - vorzeichenrichtiger betragskleinster Wert

Man **beachte**, daß `VtOP` im rufenden Programm als `EXTERNAL` vereinbart sein muß!

Nach dieser Operation liegt das Ergebnis zugleich auf jedem Prozessor vor.

Beispiel: Berechnung eines Skalarprodukts bei verteilt gespeichertem Vektor

```
  ...
DOUBLE PRECISION X(*), Y(*), HELP, SUM, DSCAPR
EXTERNAL         VDPLUS
  ...
SUM = DSCAPR (Nlocal,X,Y)
call CUBE_DOD (1,SUM,SUM,HELP,VDPLUS)
  ...
```

oder gleichzeitig zwei Skalarprodukte:

```
  ...
DOUBLE PRECISION X(*),Y(*),V(*),W(*), HELP(2), SUM(2), DSCAPR
EXTERNAL         VDPLUS
  ...
SUM(1) = DSCAPR (Nlocal,X,Y)
SUM(2) = DSCAPR (Nlocal,V,W)
call CUBE_DOD (2,SUM,SUM,HELP,VDPLUS)
  ...
```

3.2.6 `TREEUP_DOD/TREEUP_DOI` - Operationen über alle Prozessoren

Rufzeile: `call TREEUP_DOx ( N, X, Y, H, VtOP )`

Parameter:

`N` - Länge der Operandenfelder je Prozessor (oft ist `N=1`)
`X` - Resultatvektor auf Prozessor 0
- Hilfsvektor für Zwischenrechnung auf anderen Prozessoren
`Y` - Operanden-Vektor des aktuellen Prozessors
`H` - Hilfsvektor, der zur Prozessorkommunikation erforderlich ist
`VtOP` - Name einer Vektoroperation (siehe Abschnitt 2.2.1) mit der Rufzeile: `call VtOP (N,X,ix,Y,iy,Z,iz)`

Die Vektoren sind vom Typ `DOUBLE PRECISION` für `x = D` bzw. `REAL` oder `INTEGER` für `x = I`

Funktion: Jeder Prozessor hat lokal Daten berechnet (Vektor `Y`), die nun über alle Prozessoren mittels einer (kommutativen) Operation `VtOP` verknüpft werden. Dabei steht „`t`“ für `D` oder `R` oder `I`, entsprechend dem Typ der Operanden, und „`OP`“ für eine der (komponentenweise auszuführenden) Operationen (siehe 3.2.5).

Man **beachte**, daß `VtOP` im rufenden Programm als `EXTERNAL` vereinbart sein muß!

Im Gegensatz zur Operation `CUBE_DOx` liegt das Ergebnis nach dieser Operation nur auf Prozessor 0 vor. Die Nutzung dieser Routine erscheint zweckmäßig, wenn eine spezielle *Arbeitsteilung* zwischen Prozessor 0 und den anderen Prozessoren möglich ist, z.B.: während Prozessor 0 ein (kleines) Grobgitter-Gleichungssystem löst, können die anderen Prozessoren einstweilen Daten untereinander austauschen.

Beispiel: Vorkonditionierung mittels Grobgitterlöser auf Prozessor 0

```
        ...
      DOUBLE PRECISION W(*), V(*), CC(*)
      INTEGER          LCC(*), Kette(5,*)
      EXTERNAL         VDplus
        ...
      CALL TreeUp_DoD (NCrossG*Nfg,W,W,V,VDplus)
      IF (ICH .EQ. 0) THEN
        call SOLVER (NCrossG*Nfg,CC,LCC,W,W)
      ENDIF
C
      call KettAkk (Nfg,W,Kette,V)
C          an dieser Operation ist Prozessor 0 aufgrund
C          der Gebietszerlegungsstrategie nicht oder nur
C          in geringem Umfang beteiligt.
C
      NW=2*NCrossG*Nfg
      call TREE_DOWN(NW,W)
        ...
```

3.2.7 `CUBE_DIM` - Hypercube-Dimension ändern

Rufzeile: `call CUBE_DIM ( NewDim )`

Parameter:

`NewDim` - neue Dimension des (Sub-) Hypercubes, sie darf zwischen `0` und `NCUBE` liegen, wobei `NCUBE` die ursprüngliche Größe nach `TrInit` bezeichnet.

Funktion: Dieses Programm ist vor allem dazu gedacht, Testserien in kleinerem Rahmen mit unterschiedlicher Prozessoranzahl bequemer durchführen zu können, indem man beispielsweise in einer Partition von 16 Prozessoren nacheinander mit 1, 2, 4, 8 und 16 Prozessoren arbeiten kann. Im Interesse der sinnvollen Auslastung des Parallelrechners sollte diese Funktion jedoch nicht innerhalb größerer Prozessor-Partitionen mißbraucht werden.

Achtung! – Mit dem Aufruf von `CUBE_DIM` werden auch die für die Topologie wesentlichen Variablen (vgl. S. 10) `NCUBE`, `NPROC` usw. neu belegt (`ICH` wird nicht verändert). Man sollte also wenigstens den ursprünglichen Wert von `NCUBE` im Programm sichern, wie es das folgende Beispiel zeigt. Die Größen zur Bestimmung des Prozessorrings werden ebenfalls an den Sub-Cube angepaßt (Reihenfolge entspricht dem Gray-Code).

Beispiel: Vergleichende Lösung einer Aufgabe mit unterschiedlicher Prozessoranzahl, Eingabe der Cube-Dimension im Dialog.

```
        ...
       include 'trnet.inc'
        ...
       call TrInit(0)
       NCUBE_orig = NCUBE
 1     CONTINUE
       IF (ICH .EQ. 0) THEN
         write (*,'(A,$)') ' Cube-Dimension: '
         read  (*,*)   NewDim
         IF ( NewDim .GT. NCUBE_orig ) GOTO 1
       ENDIF
       L=1
       call TREE_DOWN (L,NewDim)
       IF (NewDim .LT. 0) call TrClose
       call CUBE_DIM (NewDim)
       IF ( ICH .LT. NPROC ) THEN
          ...
 C        Aufgabe im Sub-Cube bearbeiten
          ...
       END IF
 C  Das Ruecksetzen auf die urspruengliche Groesse ist hier notwendig,
 C  damit beim erneuten Durchlauf alle Prozessoren ueber TREE_DOWN
 C  die naechste zu verwendende Cube-Dimension erfahren!
       call CUBE_DIM (NCUBE_orig)
       GOTO 1
        ...
```

3.2.8 `RING_OUT` - Ausgabe über Prozessor-Ring

Rufzeile: `call RING_OUT ( Nlocal, Xin, Help, MaxH )`

Parameter:

`Nlocal` - Länge des Datenfeldes vom aktuellen Prozessor (in Worten)
`Xin` - Datenfeld, das der aktuelle Prozessor liefert
`Help` - Hilfsfeld für die Realisierung der Kommunikation
`MaxH` - verfügbare Länge des Hilfsfeldes; muß mindestens Maximum aller Werte `Nlocal` (von allen Prozessoren) sein

Funktion: Diese Funktion sollte anstelle von `TREE_UP` verwendet werden, um größere Datenpakete von allen Prozessoren zwecks Ausgabe zu Prozessor 0 zu transportieren. Dies funktioniert leider nur sequentiell, aber der Engpaß liegt ohnehin in der Übertragungsrate zum Hostrechner, nicht im Parallelrechner.

Diese Routine darf nicht auf Prozessor 0 verwendet werden, dort muß statt dessen eine Routine `RING_RECEIVE0` insgesamt `(NPROC-1)`-mal aufgerufen werden (siehe Beispiel!). Falls die Herkunft der Daten (Prozessornummer) für die Ausgabe wichtig ist, so muß diese Information entweder im Datenpaket enthalten sein oder ist anhand der bekannten Reihenfolge der Prozessoren im Ring (Gray-Code) zu berechnen. Demnach gilt für die Prozessornummern $r =$ `ICHRING` und $c =$ `ICH` die Beziehung: $c = \mathtt{XOR}(r, r/2)$.

`RING_OUT` transportiert in Richtung von Link `Lforw`, so daß die Daten vom Prozessor `ICHRING=Nproc-1` zuerst und die von Prozessor `ICHRING=1` zuletzt bei Prozessor 0 eintreffen.

Bemerkung: Unter Parix realisiert unsere Initialisierungsroutine `TrInit` im Normalfall eine andere Numerierung der Prozessoren für die Ringkommunikation (bei der alle Links auch physisch vorhanden sind!). Um die oben erwähnte Numerierung zu erzwingen, kann man nach `TrInit` das folgende Programm rufen:

```
call CUBE_DIM (NCUBE)
```

Beispiel: Ausgabe von grafischen Koordinaten, die jeder Prozessor für sein Teilgebiet geliefert hat:

```
         ...
        DIMENSION  X(3,*), HELP(*)
        include 'trnet.inc'
         ...
        IF ( ICH .GT. 0 ) THEN
          call RING_OUT ( 3*Nlocal, X, Help, MaxH )
        ELSE
C         auf Prozessor 0: zuerst eigene Daten ausgeben
          call Ausgabe ( Nlocal, X, ... )
C         dann die von allen anderen Prozessoren
          DO  I=2,NPROC
            call RING_RECEIVE0 ( N, Help, MaxH )
            call Ausgabe ( N/3, Help, ... )
          END DO
        END IF
```

3.2.9 `RING_FORW/RING_BACK` - Versenden von Daten im Prozessorring

Rufzeile: `call RING_FORW/RING_BACK ( NS, XS, NR, XR )`

Parameter:

- `NS` - Länge des zu sendenden Datenfeldes (in Worten)
- `XS` - Datenfeld, das der aktuelle Prozessor an seinen Ringnachbarn zu senden hat
- `NR` - Länge des empfangenen Datenfeldes
- `XR` - Feld, auf dem die Daten vom anderen Ringnachbarn empfangen werden können

Funktion: Mit dieser Funktion kann ein zyklischer Datenaustausch zwischen den Prozessoren in der Reihenfolge der Ringnumerierung (`ICHRING`) realisiert werden. Dabei sendet `RING_FORW` in Richtung zur höheren Nummer und `RING_BACK` zur niedrigeren Prozessornummer, jeweils modulo `NPROC`. Mit dem einmaligen Aufruf einundderselben Routine auf **allen** Prozessoren sendet jeder Prozessor seine Daten an den Nachfolger bzw. Vorgänger und empfängt zugleich Daten vom Vorgänger bzw. Nachfolger. Zu beachten ist, daß die beiden Felder `XS` und `XR` nicht identisch sein dürfen, sonst könnten die eigenen Daten teilweise überschrieben worden sein, bevor sie an den Nachbarn gesendet werden.

Beispiel: Eine Aufgabe, bei der an Gebietsgrenzen in jedem Iterationsschritt ein Datenfluß nur in eine Richtung erfolgt.

```
       ...
      DIMENSION X(N,2), Y(*)
       ...
      myData=1

C  Iterationszyklus:
       ...
C      lokale Berechnungen auf Y(*)
C      auf X(*,myData) entstehen Randdaten, die an den
C      naechsten Prozessor (ICHRING+1) gesendet werden sollen
       ...
      newData=3-myData
      call RING_FORW ( N,X(1,myData),NN,X(1,newData) )
      myData=NewData
C      eventuell Auswertung der Daten vom Vorgaenger,
C      die nun auf X(*,myData) stehen
       ...
C  Ende des Iterationszyklus
       ...
```

3.2.10 Hilfsprogramme für Zeitmessungen

Leider ist auch die Bestimmung der Rechenzeit auf jeder Hardware und unter jeder Software immer wieder anders zu realisieren. Auch hier soll (wie bei `SEND` und `RECEIVE`) ein Unterprogramm als Standard angeboten werden, das leicht portierbar erscheint und somit Grundlage für andere Routinen sein kann:

```
REAL FUNCTION SECNDS (START)
```

Die Funktion liefert die Zeitdifferenz zum Parameterwert `START` als `REAL`-Wert in Sekunden. Es wird in der Regel die sogenannte *„Liegezeit“* bestimmt (nicht die reine CPU-Zeit), was bei parallelen Programmen auch die Wartezeiten bei Kommunikationen einschließt. Die Zeit für ein Programm wird also z. B. nach folgendem Prinzip gemessen:

```
time = SECNDS ( 0.0 )
call  UP_0815 ( ... )
time = SECNDS ( time )
```

In der vorliegenden Kommunikations-Bibliothek sind alle Unterprogramme der untersten Ebene (`SEND_CHAN_0`, `RECV_CHAN_0`, `SEND_NODE_1`, `RECV_NODE_1`) in einer Testversion enthalten, bei der die Kommunikationszeiten gemessen und intern aufsummiert werden, so daß eine spätere Auswertung möglich ist.

Um diese Möglichkeit zu nutzen, die zugleich auch die Auswertung und Ausgabe der entsprechenden Werte von allen Prozessoren einschließt, können wir folgende Routinen empfehlen:

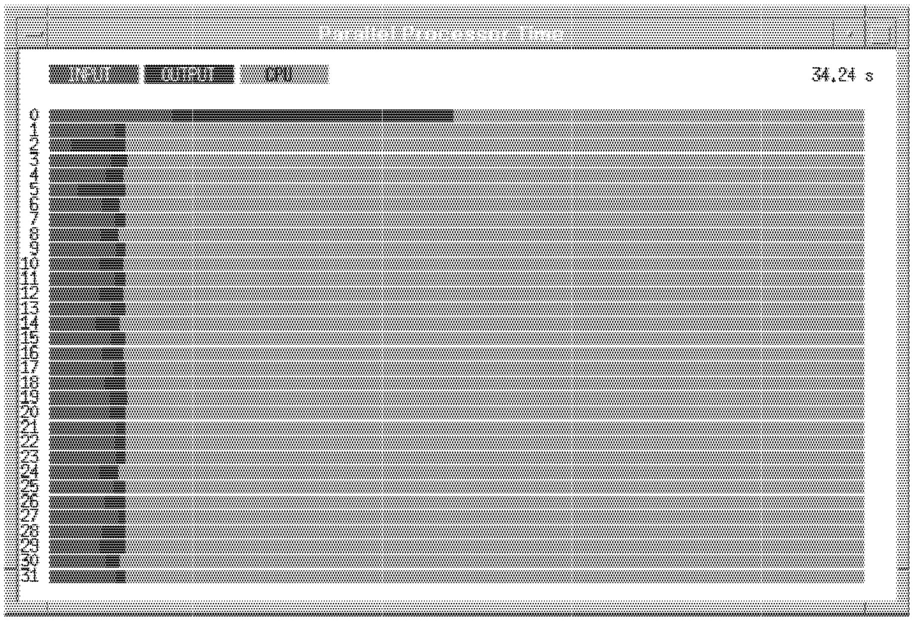
`INIT_TIME`	- Beginn der globalen Zeitmessung (kann während eines Programmlaufs mehrfach erfolgen)
`GET_TIMES(H)`	- Bestimmt die verbrauchte Zeit seit `INIT_TIME` und sammelt diese Werte von allen Prozessoren ein (benötigt ein Hilfsfeld von 6∗`NPROC` Worten)
`WRITE_TIMES(H)`	- Gibt die gemessenen Zeiten aus dem mit `GET_TIMES` belegten Hilfsfeld in einer Tabelle aus (nur auf Prozessor 0 aufrufen!)
`TIME_GRAF(H)`	- Grafische Ausgabe der gemessenen Zeiten ; - diese Routine befindet sich in der Bibliothek `libGraf.a` !

Die durch `WRITE_TIMES` ausgegebene Tabelle enthält für jeden Prozessor die folgenden Einträge:

- Prozessornummer `ICH`
- unter Parix die physischen x, y, z-Koordinaten des Prozessors im Parix-Gitter (sonst uninteressant)
- Kommunikationszeiten für `RECV_CHAN` seit letztem `Init_Time` in Sekunden und in Prozent der gesamten Laufzeit
- Kommunikationszeiten für `SEND_CHAN` (in analoger Weise)
- Gesamtlaufzeit, ebenfalls gemessen seit letztem `Init_Time`

Für große Prozessoranzahlen wurde bei der Ausgabe der Tabelle lediglich aus Gründen der Überschaubarkeit am Bildschirm eine Beschränkung auf maximal 16 Einträge vorgenommen.

Die Ausgabe durch `TIME_GRAF` hat folgende Form:



In einem separaten Grafikfenster wird ein Balkendiagramm dargestellt, in dem für jeden Prozessor die gemessenen Werte für Input (`recv`, *rot*), für Output (`send`, *blau*) und die als Differenz zur Gesamtlaufzeit ermittelte CPU-Zeit (*grün*) angezeigt werden. Damit läßt sich die Verteilung der Kommunikations- und Rechenzeiten auf allen Prozessoren eindrucksvoll veranschaulichen.

3.3 Einbinden der Bibliothek in C-Quellen

Um die in Fortran geschriebenen Kommunikationsroutinen innerhalb von C-Programmen zu verwenden, sind die bereits in 2.5.4 genannten Bedingungen bezüglich der Unterprogrammnamen und der Parameterübergabe einzuhalten.

Die für C erforderlichen Vereinbarungen sind in einem Headerfile `trnet.h` (in Analogie zu `trnet.inc` bei Fortran) zusammengefaßt, das im Verzeichnis `/usr/global/lib/parix` zu finden ist. Damit stehen auch die in 3.1 genannten Variablen aus den Fortran-Common-Blöcken innerhalb des C-Programms zur Verfügung. Sie sind folgendermaßen zu verwenden:

Fortran	Entsprechung in C
`NCUBE`	`trnet_.ncube`
`ICH`	`trnet_.ich`
`NPROC`	`trring_.nproc`
`ICHRING`	`trring_.ichring`
`LFORW`	`trring_.lforw`
`LBACK`	`trring_.lback`

Ein Beispiel zeigt die typische Vorgehensweise:

```
#include <stdio.h>
#include "/usr/global/lib/parix/trnet.h"
#include "/usr/global/lib/parix/vbasmod.h"

#define   LANG     10000
#define   DL       sizeof(double)

void main()
{
   int          istart,  nlocal = LANG,
                 i, nsum, lang, size,
                 mapping = 0;

   double        x[LANG],  y[LANG],
                 help,
                 sum = 0,

   f_init();                  /* Init. Fortran-Laufzeitsystem */

   trinit_(&mapping);         /* Initialisiere Hypercube      */

   nlocal = LANG;
   nsum   = 1;
   lang   = LANG;
   istart = trnet_.ich * nlocal + 1;
   for (i = 0; i < nlocal; i++)
   {                          /*  Vektor lokal "ausrechnen"   */
       x[i] = istart;         /*  istart ist globaler Index   */
       y[i] = 1.0 - istart;
       istart++;
   }
   sum = dscapr_(&lang, x, y); /*   lokales Skalarprodukt     */
                               /*   dann: Cube-Summe          */
   cube_dod_(&nsum, &sum, &sum, &help, vdplus_);

   if (trnet_.ich == 0)
      printf("result : %lf\n", sum);

   trclose_();                 /* Paralleles Programm beenden */
   f_exit();
}
```

3.4 Verfügbarkeit der Bibliothek

Die Bibliothek steht innerhalb der Fakultät für Mathematik der TU Chemnitz-Zwickau für Transputersysteme unter Parix mit dem Namen

`libCubecom.a` im Verzeichnis `/usr/global/lib/parix`

zur Verfügung, kann also folgendermaßen eingebunden werden:

```
f77.px ...-L/usr/global/lib/parix -lCubecom -lvbasmod
```

In der Regel wird die Bibliothek `vbasmod` ebenfalls erforderlich sein.

Darüber hinaus gibt es die Bibliothek auch für Workstation-Cluster unter PVM (= **P**arallel **V**irtual **M**achines, [7]) sowie für Einzelprozessoren (PC oder Workstation). Da PVM auch unter LINUX anwendbar ist, ergeben sich auch dort interessante Testmöglichkeiten.

Unter PVM wird innerhalb der Initialisierungsroutine (`TRINIT`) die aktuelle Hypercube-Dimension `NCUBE` im Dialog erfragt, erst danach werden die Prozesse mit `ICH`> 0 gestartet, wobei das Verteilungsprinzip der Prozesse auf die einzelnen Maschinen des vorher definierten Workstation-Clusters in der Macht des PVM-Deamons (`pvmd`) liegt. Es sind auch mehrere Prozesse auf derselben Maschine möglich, solange der Speicher reicht.

Die gegenwärtige Realisierung