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 $\lvert$`Y(i)/Z(i)`$\rvert \leq$ `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 $\texttt{NPROC} = 2^{\texttt{NCUBE}}$ 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` $= 0 \Rightarrow$ `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: $c = \texttt{XOR}(r, r/2)$.

`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 einunderselben 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:

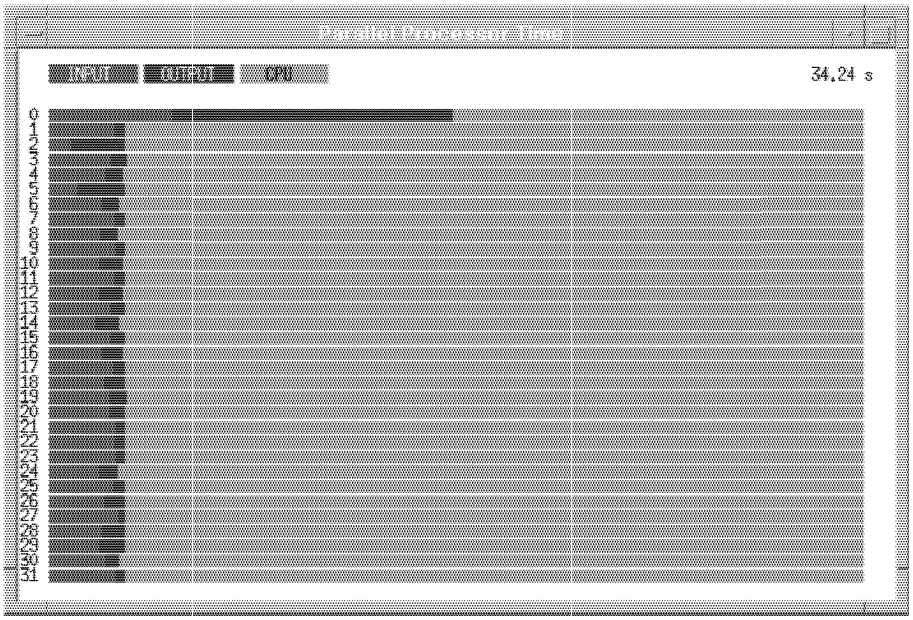
`INIT_TIME`	- Beginn der globalen Zeitmessung (kann während eines Programmlaufs mehrfach erfolgen)
`GET_TIMES(H)`	- Bestimmt die verbrauchte Zeit seit `INIT_TIME` und sammelt diese Werte von allen Prozessoren ein (benötigt ein Hilfsfeld von 6∗`NPROC` Worten)
`WRITE_TIMES(H)`	- Gibt die gemessenen Zeiten 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 für PVM Version 3.2 ermöglicht auch den Test der Programme auf nur einer Maschine, ohne daß der Deamon von PVM läuft.

Eine PC-Version erscheint zunächst widersinnig im Zusammenhang mit parallelen Algorithmen. Eine Besonderheit der Kommunikationsroutinen liegt aber gerade darin, daß sie auch für `NCUBE = 0, NPROC = 1, ICH = 0` problemlos aufgerufen werden können. Daher kann ein für Parallelrechner geschriebenes (bezüglich der Prozessoranzahl skalierbares) Programm bei Bedarf auch auf einem beliebigen anderen Rechner als Ein-Prozessor-Variante getestet werden.

3.5 Zur Wirkungsweise der Kommunikationsroutinen

Alle im Abschnitt 3.2 genannten Kommunikationsroutinen beruhen auf der Hypercube-Architektur des Parallelrechners und der damit verbundenen einfach beherrschbaren Nachbarschaftsverhältnisse der Prozessoren (siehe [5, 6]).

Zur Wiederholung: Die Prozessoren eines Hypercubes der Dimension n erhalten Nummern von 0 bis $p-1$ (mit $p = 2^n$). Zwei Prozessoren sind genau dann über Link k ($k = 1, \ldots, n$) miteinander verbunden, wenn ihre Nummern **genau** in Bitposition k (von rechts gezählt) verschieden sind, d. h. die logische Verknüpfung beider Nummern mittels „XOR“ (exklusives Oder) liefert 2^{k-1}. Daher kann Prozessor `ICH` die Nummer jedes seiner Nachbarn anhand der Link-Nummern bestimmen:

$$\texttt{NACHBAR}(k) = \texttt{XOR}(\texttt{ICH}, \texttt{ISHFT}(1, k-1)),$$

wobei `ISHFT` die Standardfunktion für Bitverschiebung ist. Allerdings wird die eigentliche Prozessornummer kaum benötigt. Zur Programmierung reichen im allgemeinen die Linknummern in Verbindung mit der **eigenen** Prozessornummer aus, wenn der Hypercube nach dem genannten Prinzip erst einmal aufgebaut ist.

Zum Beispiel hat die Routine `TREE_DOWN` (Abschnitt 3.2.1) folgendes Aussehen:

```
SUBROUTINE TREE_DOWN (N,X)
INTEGER    X(N)
INCLUDE 'trnet.inc'

K=1
IF (ICH .EQ. 0) THEN
  L=NCUBE
ELSE
  L=0
  DO WHILE ( .NOT. BTEST(ICH,L) )
    L=L+1
  ENDDO
  CALL Recv_Chan ( N, X, L+1 )
ENDIF
IF (L .GT. 0) THEN
  DO K=L,1,-1
    CALL Send_Chan ( N, X, K )
  ENDDO
ENDIF
RETURN
END
```

Prozessor `0` beginnt mit der höchsten Link-Nummer und sendet der Reihe nach über alle Links. Die anderen Prozessoren erkennen an ihrer eigenen Nummer `ICH`, über welche Link-Nummer sie zunächst die Daten einzulesen haben, um sie dann über alle niedrigeren Link-Nummern weiterzugeben. Diese erste Link-Nummer entspricht der Position des letzten Nicht-Null-Bits in `ICH`.

Eine der wichtigsten Kommunikationsroutinen ist die Operation `CUBE_DOx` zur Realisierung globaler Operationen über alle Prozessoren (Abschnitt 3.2.5). Sie wird im folgenden in der Version wiedergegeben, die für einen physisch vollständigen Hypercube am besten geeignet erscheint, bei dem zu jedem Zeitpunkt jeweils eine Hälfte der Prozessoren mit der anderen Hälfte zugleich kommunizieren kann (volle Kommunikationsbandbreite).

```
SUBROUTINE Cube_DoD (N,X,Y,H,VDop)
DIMENSION  X(1),Y(1),H(1)
INCLUDE 'trnet.inc'
External VDop

Lng=2*N
CALL VDcopy ( N, X, 1, Y, 1 )
DO  L=1,NCUBE
  IF ( Btest(ICH,L-1) ) THEN
    CALL  Recv_Chan_0 (Lng,H,L)
    CALL  Send_Chan_0 (Lng,X,L)
  ELSE
    CALL  Send_Chan_0 (Lng,X,L)
    CALL  Recv_Chan_0 (Lng,H,L)
  ENDIF
  CALL VDop (N,X,1,X,1,H,1)
ENDDO
RETURN
END
```

Das Programm hat offensichtlich nichts weiter zu tun als der Reihe nach über alle Links die *Zwischenergebnisse* mit dem jeweiligen Nachbarn auszutauschen und die als Parameter angegebene Operation `VDOP` auf die beiden Operanden anzuwenden.

Die `IF`-Anweisung im Inneren der Schleife dient dem verklemmungsfreien Datenaustausch zweier Prozessoren. Die über Link `L` verbundenen Prozessoren unterscheiden sich in ihrer Prozessornummer genau in diesem einen Bit und können sich daher einigen, wer zuerst senden bzw. empfangen soll. Diese Anweisung entspricht genau dem anfangs erwähnten Unterprogramm `EXCHNG`.

Für den Fall, daß die Anzahl der physisch vorhandenen Links kleiner ist, als die für den Hypercube benötigte Anzahl (*virtuelle Links*), kommt es bei dieser Realisierung im allgemeinen zu einer gegenseitigen Behinderung, wenn alle Prozessoren zugleich Daten austauschen wollen. Hier ist eine andere Implementierung effektiver, bei der zunächst kein *Austausch* zwischen den beiden Prozessoren erfolgt. Statt dessen werden die Zwischenergebnisse nur (wie bei `TREE_UP`) in Richtung zum Prozessor `0` weitergegeben und das Endergebnis von Prozessor `0` mittels `TREE_DOWN` wieder verteilt. Dabei entsteht theoretisch der gleiche Zeitaufwand für die Kommunikation, aber die Gesamtbelastung des Netzwerks wird geringer. Das spielt unter Parix zum Beispiel bei mehr als 16 Prozessoren schon eine Rolle.

Ein derartiges Programm steht unter dem Namen `TREE_DOD` mit der gleichen Rufzeile wie `CUBE_DOD` zur Verfügung, so daß lediglich durch Umbenennung das eine Programm das andere ersetzen kann. Für die jeweilige Hardware sollte also immer das effektivere unter dem Namen `CUBE_DOD` in der Bibliothek zu finden sein.

Das Programm `TREE_DOD` ist damit identisch mit dem Aufruf der beiden Programme `TREEUP_DOD` und `TREE_DOWN`.

4 Ein einfaches Grafikinterface für Fortran

4.1 Zweck der Bibliothek

Zielstellung bei der Entwicklung dieses Unterprogrammpaketes war es, einfache Grafikausgaben auf UNIX–Workstations unter FORTRAN zu realisieren. Ebenfalls ist dieses Interface vom Parallelrechner (Parix) aus nutzbar. Der Anwendungsprogrammierer soll nicht mehr direkt mit den Rufen der `Xlib`–Bibliothek konfrontiert werden und muß in seinen Programmen auch nicht das ereignisgesteuerte Programmiermodell berücksichtigen, wie es sonst bei X–Anwendungen üblich ist ([1], [3], [4]). Es können mehrere Fenster zur Grafikdarstellung eröffnet werden. Obwohl der Schwerpunkt auf passiver Grafik liegt, werden auch Routinen zur Eingabe von Zahlen und Zeichenketten oder zur Tastatur– und Mausabfrage bereitgestellt.

Das Objektfile `ggraph_x11.o` stellt eine Sammlung von Unterprogrammen zur Darstellung von Grafik unter X–Windows dar.

Diese Unterprogramme können in eigene FORTRAN–Programme eingebunden werden. Dazu ist es lediglich nötig, das Objektfile `ggraph_x11.o` an das eigene Programm zu linken. Zusätzlich muß die X11–Bibliothek mit eingebunden werden.

Weitere Hinweise findet man dazu in einem `README`-File. Das Objektfile `ggraph_x11.o` ist auch in der Bibliothek `libGraf.a` enthalten, die unter `/usr/global/lib/parix` zu finden ist.

Bsp.:
```
f77 myprog.f ggraph_x11.o -L... -lX11 -o myprog          oder
f77 myprog.f -L/usr/global/lib/parix -lGraf -L... -lX11 -o myprog
```

4.2 Bemerkungen zur Funktionsweise

Das Grafikpaket basiert auf den Rufen des X–Windows–Systems, und müßte sich deshalb leicht auf alle UNIX–Workstations portieren lassen, die eine grafische Benutzeroberfläche haben (bis jetzt getestet auf SUN, HP 9000, Parsytec-Cluster (Parix), PC (Linux)).

Das fertige Programm verhält sich wie eine echte X–Anwendung und kann so auch dessen Vorzüge nutzen. Durch das Setzen der `DISPLAY`–Umgebungsvariablen oder durch explizite Angabe im Programm lassen sich die Grafikausgaben auf andere Rechner umlenken (Bem.: `-display host:0` in der Kommandozeile funktioniert allerdings nicht). Die Kenntnis des ereignisgesteuerten Programmiermodells ist hier im Gegensatz zum Programmieren von X–Anwendungen nicht notwendig; das Programmiermodell ist prozedurorientiert.

Im Gegensatz zur Vorgängerversion wird direkt die X11–Bibliothek genutzt. Es gibt keinen `gserver`–Prozeß mehr. Dadurch ergab sich eine wesentliche Beschleunigung der Grafikausgabe. Diese ist unter Parix (auf Transputersystemen) und natürlich unter PVM (auf Workstation-Clustern) nutzbar. Einige Parallelrechner (NCUBE, KSR) unterstützen Grafik in dieser Form bisher leider nicht.

Um das ereignisgesteuerte Programmiermodell zu umgehen, werden alle Fensterinhalte in internen Kopien zwischengespeichert (Backing–Store Attribut). Diese benötigen relativ viel Speicher auf dem X–Server. Deshalb ist es ungünstig, sehr große Fenster anzulegen (besonders bei den X–Terminals mit relativ wenig Speicher).

4.3 Beschreibung der Unterprogramme in `ggraph`

Die Koordinaten beziehen sich stets auf die linke obere Ecke eines Fensters und beginnen bei Pixel (0,0).
Alle nicht weiter gekennzeichneten Parameter sind vom Typ `INTEGER*4`. Zeichenketten können als Konstanten oder als Zeichenkettenvariablen übergeben werden.
Die Namensgebung der UP–Rufe ist an die Namen der entsprechenden X–Rufe angelehnt.

`call GSetDisplay ( displayname )`
Die (erst nach `GSetDisplay` aufzurufende!) Funktion `GOpenServer` nutzt anstelle der `DISPLAY`-Variablen das angegebene Display.
`displayname` ist eine Zeichenkette der Form `"hostname:0"`.

`call GOpenServer ( ierr )`(*)
Es wird Verbindung zum X–Server aufgenommen. Dieses Unterprogramm **muß** einmal vor der Nutzung weiterer Grafikroutinen gerufen werden (außer `GSetDisplay`). Bei erfolgreicher Ausführung besitzt `ierr` den Wert 0, im Fehlerfall (z.B. wegen falscher `DISPLAY`-Variablen) den Wert `-1`.

`call GCloseServer`
Alle Grafik wird beendet. Die vom X–Server verwalteten Ressourcen werden freigegeben.

`call GOpenWin ( iwin, ix, iy )`(*)
Es wird ein leeres Fenster der Größe `ix * iy` erzeugt. Als Ergebnis wird die Kennzahl `iwin` zurückgegeben, die bei allen weiteren Rufen anzugeben ist. Die Position des Fensters bezüglich des gesamten Bildschirmes wird vom Windowmanager festgelegt. Die angegebene Größe wird als Minimalgröße betrachtet (durch Benutzer nicht zu verkleinern).

`call GOpenWinRel ( iwin, ix, iy, jx, jy )`(*)
Es wird wie bei `GOpenWin` ein Fenster erzeugt. Im Gegensatz kann jedoch die Position des Fensters bezüglich des gesamten Bildschirms mit `(jx,jy)` (linke obere Ecke) festgelegt werden.

`call GOpenWinGeom ( iwin, geometry )`(*)
Es wird wie bei `GOpenWin` ein Fenster erzeugt. Die Größe und Position werden in Form einer Zeichenkette angegeben wie sie bei X-Anwendungen üblich ist:
`"<breite>x<hoehe>±<xpos>±<ypos>"`. – Es sollten stets alle 4 Werte angegeben werden.

`call GCloseWin ( iwin )`
Das Fenster `iwin` wird geschlossen und vom Bildschirm entfernt. Die Kennzahl `iwin` ist danach als ungültig zu betrachten.

`call GStoreName ( iwin, title )`
Die Zeichenkette `title` wird als Titel des Fensters `iwin` angezeigt.

`call GVisualClass ( iclass )`(*)
Es wird die Visual Class zurückgegeben. Damit läßt sich feststellen, ob der angeschlossene Bildschirm s/w, Graustufen oder Farbdarstellung erlaubt. Die zurückgegebene Zahl hat folgende Bedeutung:

0	—	StaticGray	3	—	PseudoColor
1	—	GrayScale	4	—	TrueColor
2	—	StaticColor	5	—	DirectColor

Zur genauen Bedeutung dieser Werte sei auf das X–Manual verwiesen.

`call GGetSize ( iwin, ix, iy )`(*)
Es werden die Ausmaße des Fensters `iwin` in `ix` und `iy` zurückgegeben. Sofern das Fenster manuell vergrößert/verkleinert wurde, gibt `GGetSize` die tatsächliche Größe, nicht die von `GOpenWin` zurück.

`call GResize ( iwin, ix, iy )`
Die Größe des Fenster `iwin` wird geändert. Danach ist das Fenster leer und man muß die gesamte Grafik neu zeichnen.

`call GSync`
Alle X–Puffer werden geleert. Dieses Unterprogramm sollte z.B. nach der Ausführung einer Reihe von Grafikoperationen gerufen werden (bevor die Anwendung Rechnungen durchführt oder auf eine Eingabe wartet). Damit wird sichergestellt, daß alle ausgeführten Operationen auch wirklich zu diesem Zeitpunkt auf dem Bildschirm sichtbar sind.

`call GRaiseWin ( iwin )`
Das Fenster `iwin` wird, sofern es von anderen Fenstern überdeckt wird, nach oben geholt. Achtung: Damit wird dieses Fenster noch nicht automatisch fokussiert (d.h. aktiviert). Das ist insbesondere bei Tastatureingaben zu beachten.

`call GMoveWin ( iwin, jx, jy )`(*)
Das Fenster `iwin` wird auf die Position `(jx,jy)` bzgl. des gesamten Bildschirms verschoben.

`call GCreatePixmap ( ipix, iw, ih )`(*)
Es wird ein Pufferspeicher angelegt, der in der Lage ist, einen Bereich der Größe `(iw,ih)` abzuspeichern. Die Kennzahl `ipix` wird zurückgegeben.

`call GFreePixmap ( ipix )`
Der Pufferspeicher `ipix` wird freigegeben.

`call GGetPixmap ( iwin, ipix, ix, iy, iw, ih )`
Es wird der Bereich `(ix,iy,iw,ih)` vom Fenster `iwin` in den Pufferspeicher `ipix` kopiert.

`call GPutPixmap ( iwin, ipix, ix, iy, iw, ih )`
Aus dem Pufferspeicher `ipix` wird ein Bereich der Größe `(iw,ih)` zum Fenster `iwin` an die Stelle `(ix,iy)` kopiert.

`call GClearWin ( iwin )`
Das Fenster `iwin` wird mit der durch `GSetforeground` gesetzten Farbe gefüllt.

`call GDrawPoint ( iwin, ix, iy )`
Es wird ein Punkt im Fenster `iwin` bei `ix` und `iy` in der aktuellen Vordergrundfarbe gesetzt.

`call GDrawPoints ( iwin, n, points )`
Es werden `n` Punkte gezeichnet, deren Koordinaten sich im Feld `points` befinden. Dieses Feld ist von Typ `INTEGER*2` und hat die Länge 2*n (jeweils x– und y–Koordinaten abwechselnd).

`call GDrawLine ( iwin, ix1, iy1, ix2, iy2 )`
Eine Linie wird von `(ix1,iy1)` nach `(ix2,iy2)` gezeichnet.

`call GDrawLines ( iwin, npoints, points )`
Es wird ein Polygonzug im Fenster `iwin` gezeichnet. Dazu sind `npoints` Punkte gegeben, deren Koordinaten `(x,y)` aufeinanderfolgend im Feld `points (INTEGER*2)` abgespeichert sind. `points` hat also mindestens die Dimension `2*npoints`.

`call GDrawRectangle ( iwin, ix, iy, iwidth, iheight )`
Es wird ein Rechteck im Fenster `iwin` gezeichnet. `ix, iy` geben die linke obere Ecke an, `iwidth, iheight` die Breite und Höhe.

`call GDrawArc ( iwin, ix, iy, iwidth, iheight, ialpha, ibeta )`
Es wird ein Ellipsenbogen gezeichnet. `ix, iy, iwidth, iheight` sind die Koordinaten des umschreibenden Rechtecks der gesamten Ellipse. Der Bogen wird aber nur beginnend beim Winkel `ialpha` mit einer Größe `ibeta` gezeichnet. Die Winkel werden in Grad * 64 angegeben. Genaueres siehe X–Manual.

`call GDrawCircle ( iwin, ix, iy, ir )`
Es wird ein Kreis mit dem Mittelpunkt `(ix,iy)` und dem Radius `ir` gezeichnet.

`call GFillRectangle ( iwin, ix, iy, iwidth, iheight )`
Es wird ein mit der Vordergrundfarbe gefülltes Rechteck gezeichnet.
Siehe `GDrawRectangle`.

`call GFillArc ( iwin, ix, iy, iwidth, iheight, ialpha, ibeta )`
Es wird ein mit der Vordergrundfarbe gefülltes Tortenstück gezeichnet. Siehe `GDrawArc`.

`call GFillCircle ( iwin, ix, iy, ir )`
Es wird ein mit der Vordergrundfarbe gefüllter Kreis gezeichnet. Siehe `GDrawCircle`.

`call GFillPolygon ( iwin, npoints, points, mode )`
Es wird ein mit der Vordergrundfarbe ausgefüllter Polygonzug gezeichnet. Die Parameter `iwin, npoints, points` haben die gleiche Bedeutung wie bei `GDrawLines`. Dabei wird automatisch der letzte Punkt mit dem ersten Punkt verbunden. Der Parameter `mode` kann folgende Werte besitzen:

0 — Complex (d.h. auch überschlagene Polygonzüge)
1 — Convex
2 — Nonconvex

`call GSetLineAttrib ( iwin, iwidth, istyle, icapstyle, joinstyle )`
Beim Zeichenen von Linien wird die Linienstärke auf `iwidth` Pixel gesetzt. `iwidth`=0 bedeutet auch 1 Pixel Linienstärke. `istyle` kann folgende Werte annehmen:
0 — durchgehende Linie
1 — gestrichelte Linie (nur Vordergrundfarbe)
2 — gestrichelte Linie (Vorder– / Hintergrundfarbe abwechselnd)
Zur genauen Bedeutung von `icapstyle, joinstyle` sei auf das Xlib–Manual verwiesen. (Beide Parameter bestimmen die Darstellung der Enden bzw Verbindungspunkte von Linien und können im Zweifelsfall Null gesetzt werden.)

`call GSetDrawmode ( iwin, mode )`
Gibt an, wie die Zeichenoperation durchgeführt werden soll, z.B. mode =3 (**copy**), =6 (**xor**) oder =1 (**and**) Im Normalfall wird direkt gezeichnet (**copy**). Die entsprechende logische Verknüpfung erfolgt bitweise zwischen aktuellem Bildinhalt und zu zeichnendem Pixel (dadurch wird ein neuer Farbindex *„ausgewürfelt“*, also keine nachvollziehbare Kombination von *Farbwerten*). Eine sinnvolle, vom Normalfall abweichende Darstellungsform ist **xor**, wobei jeweils die zweite Ausgabe des gleichen Objekts dieses wieder löscht. Allerdings lassen sich kaum allgemeine Aussagen machen, in welcher Farbe das Objekt nach der ersten Ausgabe zu sehen ist.

`call GSetForeground ( iwin, icolor )`
Zum Zeichnen im Fenster `iwin` wird die angegebene Farbe benutzt. `icolor` muß eine gültige Farbkennzahl sein.

`call GSetBackground ( iwin, icolor )`
Die Backgroundfarbe im Fenster `iwin` wird gesetzt. `icolor` muß eine gültige Farbkennzahl sein. Diese Farbe wird bei `GDrawImageString`, `GDrawInt,...` für den Hintergrund benutzt.

`call GParseColor ( ir, ig, ib, name )`$^{(*)}$
Es wird die Zeichenkette `name` in einer internen Datenbasis gesucht und der dazugehörige Farbeintrag ermittelt. Als Ergebnis werden die Rot–Grün–Blau–Anteile in `ir, ig, ib` zurückgegeben. Eine lesbare Kopie dieser Datenbasis befindet sich meist unter `/usr/lib/X11/rgb.txt`.

`call GAllocColor ( ir, ig, ib, icolor )`$^{(*)}$
Es wird in der Farbpalette ein neuer Farbeintrag eingerichtet. Die neue Farbe wird durch `ir, ig, ib` beschrieben (Wertebereich dieser Variablen $0 \ldots 2^{16}$). Zurückgegeben wird `icolor`, die dann z.B. bei `GSetForeGround` benutzt werden kann. Man beachte aber, daß die Anzahl der Farbeinträge je nach Display beschränkt sind und deshalb u.U. nicht die gewünschte Farbe eingetragen werden kann.

`call GFreeColor ( icolor )`
Die Farbkennzahl `icolor` und der dazugehörige Tabelleneintag werden freigegeben und können ab sofort nicht mehr benutzt werden. Es dürfen nur Farben freigegeben werden, die mit `GAllocColor` angefordert wurden.

`call GWhitePixel ( icolor )`$^{(*)}$
Die Kennzahl für weiß wird zurückgegeben.

`call GBlackPixel ( icolor )`(*)
Die Kennzahl für schwarz wird zurückgegeben.

`call GRainColor ( colors )`(*)
Es wird ein Feld `colors` der Länge 100 mit Farbkennzahlen gefüllt. Die Farben stellen in etwa einen Regenbogen dar; sie gehen von violett über blau, grün, gelb, orange, rot bis weiß.

`call GLoadQueryFont ( ifont, name )`(*)
In der Zeichenkette `name` ist der Name eines Zeichensatzes angegeben. Der Zeichensatz wird geladen (aber noch nicht aktiviert!) und eine Kennzahl `ifont` zurückgegeben. Die Namen aller verfügbaren Zeichensätze erhält man über das Unix–Kommando `xlsfonts`. Einen speziellen Zeichensatz kann man sich mit `xfd -fn` *fontname* ansehen. Man beachte, daß sich die Menge der verfügbaren Zeichensätze von Rechner zu Rechner unterschiedlich sind und es empfieht sich deshalb, nur „gängige“ Zeichensätze zu verwenden.

`call GSetFont ( iwin, ifont )`
Im Fenster `iwin` wird ab sofort der geladene Zeichensatz mit der Kennzahl `ifont` verwendet.

`call GUnloadFont ( ifont )`
Der Zeichensatz mit der Kennzahl `ifont` wird auf dem X–Server freigegeben und darf ab sofort nicht mehr benutzt werden.

`call GGetTextsize ( iwin, ifont, 'string', itop, ibottom, ileft, iright )`
Stellt den Platzbedarf zum Zeichnen von `'string'` bei Nutzung des Zeichensatzes `ifont` fest. Rückgabewerte sind `itop, ibottom, ileft, iright` (Anzahl der Pixel, gerechnet vom Urpsrung aus, der als Position für `GDrawString` verwendet wird).

`call GDrawString ( iwin, ix, iy, s )`
Es wird eine Zeichenkette `s` in der gerade aktuellen Schriftart geschrieben. `ix, iy` beziehen sich auf die linke untere Ecke des ersten Zeichens.

`call GDrawImageString ( iwin, ix, iy, s )`
wirkt wie `GDrawString`, jedoch wird der durch die Zeichenkette überdeckte Platz mit der Hintergrundfarbe gefüllt, wohingegen bei `GDrawString` „durchsichtig“ geschrieben wird.

`call GReadKey ( iwin, ic )`(*)
Es wird ein Zeichen von der Tastatur bezüglich des Fensters `iwin` geholt. Zur Eingabe muß also das Fenster aktiviert sein. Jedes Fenster besitzt seinen eigenen Tastaturpuffer; es werden alle Tastendrücke gepuffert, auch wenn erst mal kein `GReadKey` gerufen wird. In `ic` wird der Tastencode zurückgegeben. Falls der Tastaturpuffer leer ist, wird 0 zurückgegeben (ohne auf ein Zeichen zu warten).

`call GKeyPressed ( iwin, istatus )`(*)
`istatus` ist 1, falls der Tastaturpuffer nicht leer ist; ansonsten 0.

`call GClearKeys ( iwin )`
Löscht den Tastaturpuffer bzgl. des Fensters `iwin`.

`call GReadString ( iwin, ix, iy, s )`(*)
Es wird eine Zeichenkette s eingelesen. Vorher wird aber noch der Tastaturpuffer gelöscht. Ist die eingegebene Zeichenkette länger als `s`, so wird abgeschnitten; ist sie kürzer, dann wird sie mit Leerzeichen aufgefüllt. Das Schreiben auf dem Bildschirm geschieht sofort (in der Art von `GDrawImageString`), ein `GSync` ist nicht erforderlich. Während des Eingabevorgangs kann die Eingabe mit der Backspacetaste korrigiert werden; Abschluß mit der Return–Taste.

`call GMouse ( iwin, ix, iy, ibuttons )`(*)
Es wird die Position und der Status der Maustasten ermittelt. Die Position bezieht sich stets auf das angegebene Fenster und kann somit auch negativ werden, wenn sich die Maus links oder oberhalb vom Fenster befindet. `ibuttons` haben bei einer 3–Tasten–Maus folgende Bedeutung:

0 — keine Taste gedückt
1 — linke Taste
2 — mittlere Taste
4 — rechte Taste

Sind mehrere Tasten gedrückt, wird die Summe der Zahlen übergeben.

`call GMouseCursor ( iwin, icursor )`
Im Fenster `iwin` soll der Mauscursor die Form `icursor` annehmen. Einen Überblick über die möglichen Kennzahlen kann man sich mit dem Unix–Kommando `xfd -fn cursor` verschaffen.

`call GBell`
Ein akustisches Signal wird ausgegeben.

`call GDrawReal ( iwin, ix, iy, d, l, k )`
Die `DOUBLE PRESICION`–Zahl d wird im F–Format ausgegeben. Dabei gibt l die Gesamtlänge an, k die Anzahl der Nachkommastellen.

`call GDrawDouble ( iwin, ix, iy, d, l, k )`
Die `DOUBLE PRESICION`–Zahl d wird im E–Format ausgegeben. Dabei gibt l die Gesamtlänge an, k die Anzahl der Mantissenstellen.

`call GDrawInt ( iwin, ix, iy, d, l )`
Die `INTEGER*4`–Zahl d wird im I–Format ausgegeben. Dabei gibt l die Gesamtlänge an.

`call GReadDouble ( iwin, ix, iy, d, ierr )`(*)
Es wird eine `DOUBLE PRESICION`–Zahl d eingelesen. Die Eingabe kann in Festkommaschreibweise oder in Gleitkommaschreibweise (mit e oder E, aber nicht mit d, D als Exponenten) erfolgen. `ierr` ist 0, falls die Konvertierung erfolgreich durchgeführt werden konnte, ansonsten 1.

`call GReadInt ( iwin, ix, iy, d, ierr )`(*)
Es wird eine `INTEGER*4`–Zahl d eingelesen. `ierr` ist 0, falls die Konvertierung erfolgreich durchgeführt werden konnte, ansonsten 1.

`call GWriteKey ( iwin, ichar )`
Emuliert einen Tastendruck. Das Zeichen mit dem ASCII-Code `ichar` kann dann mittels `greadkey` eingelesen werden. Diese Funktion hat im Normalfall nicht viel Sinn. Das Zeichen sollte kein Ctrl-Zeichen sein (kleiner '␣') und muß auf der Tastatur auch vorkommen.

4.4 Fehlerbehandlung

Um das System möglichst einfach zu halten, fallen die Fehlermeldungen nur rudimentär aus. Der Programmierer hat selbst dafür zu sorgen, daß alle Parameter mit sinnvollen Werten belegt sind. Insbesondere hat er auf die korrekte Anzahl der Parameter zu achten. Einige wenige Fehlermeldungen/Warnungen werden von `ggraph` ausgegeben. Oft befindet sich hinter dem eigentlichen Fehlertext noch eine kurze englische Erläuterung. Das ist insbesondere dann der Fall, wenn es sich um Fehler der Systemfunktionen handelt.

- `ggraph_x11: ungültiger Fensterindex`
  Der Parameter `iwin` eines Unterprogrammrufes ist falsch. Man überprüfe, ob das Fenster richtig eröffnet wurde und ob `iwin` einen definierten Wert besitzt.
- `ggraph_x11: Event unbekannt, wird nicht bearbeitet, Event type=...`
  Dieser Fehler dürfte auch normalerweise nicht auftreten.
- `ggraph_x11: Zeichensatz nicht gefunden`

Die meisten anderen Fehler werden einfach ignoriert, wie z.B. Koordinaten außerhalb eines Fensters oder falsche Farben. Andere schwerwiegende Fehler, wie z.B. kein freier Speicherplatz auf dem X–Server, bewirken eine Fehlermeldung des X–Window–Systems. So eine Fehlermeldung erkennt man auch leicht als solche. Eine Weiterarbeit ist dann meist nicht mehr möglich.

4.5 Besonderheiten bei der Nutzung unter Parix

Bei der Generierung der Gafikbibliothek kann man durch Optionen angeben, ob die Unterprogramme nur von einem Prozessor gerufen werden dürfen oder ob alle Prozessoren Zeichenoperationen in das gleiche Fenster durchführen dürfen.

Im ersten Fall ist ergeben sich kaum Unterschiede zu den sequentiellen Versionen unter UNIX. Alle Rufe darf nur ein Prozessor ausführen (z.B. Prozessor 0). Das Weiterleiten der Nutzerdaten zu Prozessor 0 ist Aufgabe des Anwenders (z.B. über eine Ringstruktur, siehe 3.2.8). Alle anderen Prozessoren, die nicht `GOpenServer` gerufen haben, dürfen auch keine anderen Grafikrufe ausführen. Allerdings ist es auch möglich, daß alle Prozessoren `GOpenServer` und `GOpenWin` usw. ausführen. Dann erscheint für jeden Prozessor ein eigenes Fenster, welches vollkommen unabhängig von allen anderen ist. Diese Variante ist sicherlich nur für kleine Prozessorzahlen sinnvoll.

Im zweiten Fall ist eine spezielle Variante von `ggraph` nötig. Es wird nur ein Fenster eröffnet, aber alle Prozessoren können gleichzeitig in dieses Fenster zeichnen. Dabei ist wie folgt vorzugehen:

- Alle Prozessoren nehmen durch `call GOpenServer` Verbindung zum X–Server auf.
- Alle Prozessoren eröffnen das Fenster (`call GOpenWin`). Dabei erzeugt nur Prozessor 0 tatsächlich ein Fenster, alle anderen Prozessoren führen nur Initialisierungen durch. Weiterhin findet in `GOpenWin` eine Kommunikation statt (`Tree_Down`). Auf jedem Prozessor steht ein Handle bereit, mit dessen Hilfe nun in das Fenster gezeichnet werden kann.

- Befehle wie Farben holen oder Fonts laden können entweder alle Prozessoren gleichzeitig ausführen oder (besser!) nur ein Prozessor führt diese Operation durch und gibt mittels `Tree_Down` die Werte an die anderen Prozessoren weiter.
- Alle Prozessoren können Befehle zum Zeichnen von Grafik ausführen. Für jeden Prozessor wird ein eigener Grafikkontext verwaltet. Damit wird sichergestellt, daß jeder Prozessor unabhängig von allen anderen Farben, Strichstärken u. ä. einstellen kann.
- Funktionen, die irgendwelche Werte zurückgeben, z.B. Eingabe von Zahlen, sollte nur ein Prozessor durchführen (z.B. `ICH=0`).
- Vor Beendigung des Programms sollten alle Prozessoren durch `GCloseServer` die Grafik beenden.

Es sei noch einmal erwähnt, daß diese Mehrprozessorvariante für große Prozessoranzahlen (≥ 32) nicht effektiv ist bzw. aufgrund interner Beschränkungen von X11 nicht mehr möglich ist. Weiterhin muß beim Linken die Bibliothek zur Prozessorkommunikation mit angegeben werden (`Tree_Down` wird benötigt).

4.6 Nutzung der Unterprogramme in anderen Sprachen

C: Die Namen der Unterprogramme werden alle klein geschrieben. Alle Parameter müssen Zeiger auf den entsprechenden Datentyp sein, also `int *` bzw. `char *`. Nach einem Zeichenkettenparameter sollte dessen Länge als Wertparameter int übergeben werden. Es wird kein Funktionswert zurückgegeben (`void ...`).

C: Häufig wird vom Fortran-Compiler ein Unterstreichungszeichen am Ende des Unterprogrammbezeichners angehängt (z.B. bei Sun, Linux; nicht aber bei HP). Je nach dem müssen auch die Rufe von C aus ein Unterstreichungszeichen tragen.

C unter Parix: Generell kann der ANSI-C-Compiler benutzt werden.

HP-Pascal: Die Parameter sind vom Typ `var ... integer` bzw. `var ... string`. Alle benutzten Rufe sind in Pascal als `external C;` zu deklarieren, z.B.
`procedure GOpenWin ( var A, B, C: integer ); external C;`

4.7 Hinweise und Probleme der vorliegenden Version

- Es wird eine vollständige Kopie jedes Fensterinhaltes verwaltet. Das benötigt viel Speicherplatz auf der X–Serverseite.
- Die vorliegende Unterprogrammsammlung eignet sich im wesentlichen nur für passive Grafik und Eingaben einfachster Art. Es werden keinerlei Dialogelemente oder modale Eingabeformen zur Verfügung gestellt.
- Die Biliothek baut direkt auf den X–Rufen auf. Bei der Portierung auf andere Parallelrechner, auf denen keine X11–Bibliothek verfügbar ist, kann es zu Problemen kommen.

4.8 Host-Kommunikation, Nutzung von Netzwerkfunktionen

Die Funktionen zur Netzwerkkommunikation, wie sie in der alten Version zur Verfügung standen, sind nun hinfällig. Es werden nur die Möglichkeiten genutzt, die das X bietet. Die entprechenden Funktionen der Vorgängerversion sind in der vorliegenden Fassung von `ggraph` als Dummy–Routinen ausgeführt.

4.9 Ein einfaches Beispiel

```
        integer*2 fuenfeck(10)
        character*40 sread
        data fuenfeck /30, 20, 80, 30, 160, 15, 165, 80, 130, 70/
C
C-----------  Verbindung zum gserver aufbauen  -------------------
        call  GOpenServer    ( i )
        if ( i .ne. 0 ) stop
C
C-----------  Farben laden ---------------------------------------
        call  GWhitePixel    ( iwhite )
        call  GAllocColor    ( 65000, 65000, 0, iyellow )
        call  GAllocColor    ( 0, 0, 65000, iblue )
C
C-----------  Fenster eroeffen -----------------------------------
        call  GOpenwin       ( iwin1, 300, 600  )
        call  GStoreName     ( iwin1, 'Grafikausgabe' )
        call  GSetForeGround ( iwin1, iblue )
        call  GClearWin      ( iwin1 )
        call  GSetForeGround ( iwin1, iyellow )
        call  GSetBackGround ( iwin1, iblue )
C
C-----------  Einige Zeichenoperationen --------------------------
        call  GDrawLine      ( iwin1, 0, 0, 200, 200 )
        call  GDrawRectangle ( iwin1, 10, 10, 200, 100 )
        call  GDrawCircle    ( iwin1, 80, 300, 100 )
        call  FillRectangle  ( iwin1, 10, 380, 200, 70 )
        call  GFillCircle    ( iwin1, 260, 260, 48 )
        call  GFillPolygon   ( iwin1, 5, fuenfeck, 0 )
C
C-----------  Ein-/Ausgabe im Grafikfenster --------------------
        call  GSetForeGround ( iwin1, iwhite )
        call  GDrawString    ( iwin1, 10, 500, 'String-Eingabe: ' )
        call  GReadString    ( iwin1, 200, 500, sread )
        write (*,*) 'Eingabe von Fenster: ', sread
C
C-----------  Fenster zu, Server beenden  ------------------------
        call  GClosewin     ( iwin1 )
        call  GCloseServer
C
        end
```

Literatur

[1] J. Gettys and R. W. Scheifler. Xlib - C language X interface, MIT Consortium Standard, X Version 11, Release 5. First revision, Digital Equipment Corporation and Massachussetts Institute of Technology, August 1991.

[2] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans.Math.Softw.*, 5:308–325, 1979.

[3] A. Nye. *Xlib Programming Manual for Version X11.* O'Reilly & Associates, Inc., 1990.

[4] B. Rieken and L. Weiman. *Adventures in UNIX Network Applications Programming.* John Wiley & Sons, Inc., New York – Chichester – Brisbane – Toronto – Singapore, 1992.

[5] Y. Saad and M. H. Schultz. Topological properties of hypercubes. Research Report 389, Yale University, Dept. Computer Science, 1985.

[6] Y. Saad and M. H. Schultz. Data communication in hypercubes. *Journal of parallel and distributed computing*, 6 : 115–135, 1989.

[7] V. S. Sunderam. PVM: A framework for parallel distributed computing. Technical report, Oak Ridge National Laboratory, 1992. PVM Public Domain Package.

Kontaktadresse der Autoren:

Technische Universität Chemnitz-Zwickau
Fakultät für Mathematik
Reichenhainer Straße 41

PSF 964
D-09009 Chemnitz

`e-mail: pester@mathematik.tu-chemnitz.de`