Technische Universität Chemnitz
Sonderforschungsbereich 393
*Numerische Simulation auf massiv parallelen Rechnern*

Matthias Pester

# Bibliotheken zur Entwicklung paralleler Algorithmen – Basisroutinen für Kommunikation und Grafik

**Preprint SFB393/02-01**

**Preprint-Reihe des Chemnitzer SFB 393**

**SFB393/02-01** **Januar 2002**

# Inhaltsverzeichnis

# Bibliotheken zur Entwicklung paralleler Algorithmen – Basisroutinen für Kommunikation und Grafik

(Stand: Dezember 2001)

**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 and continously extended at the University of Chemnitz since the end of the eighties. 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 PARIX for transputers and PowerPC, nCube, PVM, MPI. The vector subroutines are optimized by the use of C language and unrolled loops (*BLAS1-like*). Hardware-optimized BLAS1 routines may be integrated. The paper includes hints for programmers how to use the libraries with both Fortran and C programs.

## 1 Einleitung

Die vorliegende neue Version verfolgt den Zweck, einige an der TU Chemnitz entwickelte Programmbibliotheken neben der Dokumentation für die Nutzung im eigenen Haus weiteren Interessenten nutzbar zu machen. Gegenüber früheren Fassungen der Dokumentation wurde eine Reihe von Änderungen und Ergänzungen eingearbeitet.

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 Hierarchiestufe, 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, PowerXplorer und GCPowerPLus, 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. Sie baut auf einfachen X11-Funktionen auf und ist damit ebenfalls weitgehend portabel.

Schließlich werden einige hilfreiche Programme von allgemeinem Charakter kurz vorgestellt.

# 2 Bibliothek für Vektoroperationen – vbasmod

## 2.1 Sinn und Zweck der Vektorbibliothek

Diese Bibliothek[1] 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 6.1). Die Rufzeilen der Vektorbibliothek sind allgemeiner gehalten als die Rufzeilen der (oft funktionsidentischen) BLAS1-Routinen [3]. 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 *Entrollen* der enthaltenen Schleifen (*unrolled loops, BLAS-like*) auch die Programmiersprache C und (wenn vorhanden) hardware-optimierte 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 (abhängig von Compiler- und Prozessorfähigkeiten).
3. Außer dem Bearbeiter der Vektorbibliothek braucht sich kein Anwender der Bibliothek mit Fragen der Optimierung häufig wiederkehrender Programmteile herumzuschlagen.

## 2.2 In der Vektorbibliothek enthaltene Routinen

Die in der Vektorbibliothek enthaltenen Routinen werden in vier Gruppen mit jeweils 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`[2], 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 |

[1]Die ursprüngliche Version wurde von Arnd Meyer in FORTRAN77 geschrieben. Modifikationen und Erweiterungen der Bibliothek im Hinblick auf die Optimierung der Laufzeiten erfolgten durch Gundolf Haase, Beate Junghans und Matthias Pester.

[2]Als Schrittweite sind alle ganzen Zahlen zugelassen, speziell auch Schrittweite 0 für eine skalare Konstante anstelle eines Vektors, dann aber Vorsicht mit hohen Compileroptimierungsstufen.

| | | | |
|---|---|---|---|
| `[oper]` | = | `mult` | Multiplikation |
| | = | `div` | Division |
| | = | `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` |
| | nur für `x`=I : | | |
| | = | `mod` | Modulo: `X(i):=mod(Y(i),Z(i))` |
| | = | `and` | bitweise logisches **AND** |
| | = | `or` | bitweise logisches **OR** |
| | = | `eor` | bitweise logisches **XOR** |

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)
SUBROUTINE VIand   (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIor    (N,X,ix,Y,iy,Z,iz)
SUBROUTINE VIeor   (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`) |

| | | | | |
|---|---|---|---|---|
| Operationen : | `[oper]` | = | `copy` | Kopieren |
| | | = | `chng` | Vertauschen |
| | | = | `sqrt` | Quadratwurzel |
| | `[x]` | = | `c` | **Datentypkonvertierungen:** |
| | `[oper]` | = | `IfromR` | `REAL*4` zu `INTEGER` |
| | | = | `IfromD` | `REAL*8` zu `INTEGER` |
| | | = | `RfromI` | `INTEGER` zu `REAL*4` |
| | | = | `RfromD` | `REAL*8` zu `REAL*4` |
| | | = | `DfromI` | `INTEGER` zu `REAL*8` |
| | | = | `DfromR` | `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`, [3] |

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.

[3]Statt `R` wird hier `S` verwendet wegen der Analogie zu den Namen der BLAS-Routinen

### 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. Die drei Funktionsunterprogramme `DScapr, RScapr, IScapr` liefern einen Funktionswert des entsprechenden Typs.

## 2.3 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. Dennoch kann der Anwender eine andere Version der Bibliothek benutzen, indem diese mit ihren speziellen Namen `libvbasmodf.a, libvbasmodc.a` bzw. `libvbasblas.a` eingebunden wird.

Die Quellen (FORTRAN77, C) enthalten keine Parallelrechner-spezifischen Teile und können somit auch in anderen (seriellen) Programmen angewandt werden.

| Quellen in | Sprache | Bemerkung |
|---|---|---|
| `~/libs/src/vbasmodf` | FORTRAN | Grundversion, *unrolled loops* |
| `~/libs/src/vbasmodc` | C | äquivalent zur Grundversion |
| `~/libs/src/vbasblas` | C | BLAS1-Routinen erforderlich |

In allen Fällen sind die Unterprogramme sowohl von Fortran- als auch von C-Programmen aus nutzbar (vgl. 6.2). Unter Parix war beispielsweise die C-Version für Transputer deutlich schneller als die Fortran-Variante, was aber bei Parix für PowerPC nicht mehr zutrifft.

Die BLAS-Version der Bibliothek ist im wesentlichen mit der C-Version identisch, nutzt aber intern die echten BLAS1-Routinen, wo dies (aufgrund der aktuellen Parameter) möglich ist. Dies macht nur Sinn, wenn die BLAS1-Routinen maschinennah optimiert wurden; mit Fortran-Quellen der BLAS-Routinen ist keine Leistungssteigerung gegenüber den Routinen der vorgestellten Bibliothek zu erwarten. Solche Versionen existieren z. B. für Parix auf Transputertechnik, für nCube und für HP. Bei Linux ist in `libvbasblas.a` auch die für Pentium-III optimierte BLAS-Bibliothek enthalten. Deutlich bessere Leistung gibt es aber bei den BLAS1-Routinen nur für „kurze“ Vektoren (ca. $O(10^4)$ Elemente), weil sonst Speicherzugriffe über die Cache-Leistung dominieren.

Die Programme der Bibliothek `vbasmod` erfordern selbst keinerlei Parallelität, sind aber Voraussetzung für einige der komfortableren Kommunikationsroutinen der Bibliothek `Cubecom` (Abschnitt 3).

## 2.4 Programmbeispiele

### 2.4.1 Kopieren von Vektoren

Das Verständnis der Wirkungsweise der [copy]- und [chng]-Routinen wird durch eine Betrachtung der Daten als 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 für INTEGER bzw. REAL*4 (in C mit `void*` vereinbart).

```
      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)
              ...
```

### 2.4.2 Vertauschen 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 )
```

### 2.4.3 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, oberes Dreieck zeilenweise
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))
          ia = ia+n1+1
          n1 = n1-1
20      CONTINUE
        RETURN
        END
```

# 3 Kommunikationsbibliothek – Cubecom

## 3.1 Grundlagen

Die hier vorzustellende Bibliothek[4] stellt eine weitgehend hardware-unabhängige Kommunikationsschnittstelle dar, die das Entwickeln paralleler Algorithmen unterstützt.

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[5] unterstützt, z. B.:

| | | |
|---|---|---|
| Transputer 3L : | `F77_Chan_Out_Message (...)` | `F77_Chan_In_Message (...)` |
| Helios : | `PSX_Write (...)` | `PSX_Read (...)` |
| Parix : | `Send (...)` | `Recv (...)` |
| nCube : | `nWrite (...)` | `nRead (...)` |
| PVM 3.3 : | `pvmfsend (...)` | `pvmfrecv (...)` |
| PARMACS 6.0 : | `pm_sendr (...)` | `pm_recvr (...)` |
| MPI : | `mpi_send (...)` | `mpi_recv (...)` |

wobei die Rufzeilen in jedem Fall einige – manchmal „überflüssig" anmutende – Parameter enthalten, deren Bedeutung wenig zu tun hat mit der eigentlich vorgesehenen Anwendung, eine bestimmte Menge Daten an ein bestimmtes Ziel zu transportieren, und die nur der (beabsichtigten) Universalität des jeweiligen Systems geschuldet sind.

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 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` ($= 1, \ldots, \texttt{NCUBE}$) 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 manchen 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:

---

[4] Die Routinen wurden von Arnd Meyer und Matthias Pester entwickelt. Einige Erweiterungen stammen von Matthias Bollhöfer.

[5] In der Parix-Version 1.3 für PowerPC gibt es gerade bei den Fortran-Routinen "`send`" und "`recv`" einen *Name-Clash* (gleichnamige Unterprogramme in einer C-Bibliothek für Sockets). Aus diesem Grund wurde in unsere Kommunikationsbibliothek ein Ersatz für die Original-Parix-Routinen unter den Namen "`f_send`" bzw. "`f_recv`" aufgenommen.

```
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 (*keine Nachricht ist auch eine Nachricht*).

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 aufgefhrten 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. eine der folgenden Routinen verwendet werden:

```
SUBROUTINE EXCHNG ( LinkNo, nWords, SendData, RecvData )
  oder
SUBROUTINE V_EXCHNG( LinkNo, nSend, SendData, nRecv, RecvData )
```

Diese Routine gewährleistet einen deadlock-freien Datenaustausch, ohne das rufende Programm mit diesem Problem zu belasten. Darüber hinaus kann sie z.B. bei entsprechender Basis-Kommunikationsbibliothek (MPI) effizienter implementiert sein als die beiden Einzeloperationen. Bei `V_EXCHNG` ist zu beachten, dass die Länge `nRecv` vorher nicht bekannt sein muss (wird von der Routine zurückgegeben), aber das Feld `RecvData` muss „hinreichend“ lang sein.

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 angepasste Unterprogramm

```
SUBROUTINE TRINIT ( mode )
```

wobei der `Integer`-Parameter `mode` lediglich zu Testzwecken unter Parix verwendet wurde: Im Falle `mode=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-Gitters für den Hypercube.

Für Clustersysteme unter MPI oder PVM bleibt dieser Parameter `mode` *„reserviert für Erweiterungen“*. Durch `mode=-1` wird die Ausgabe der aktuellen Prozessorzuordnung beim Aufruf von `trinit` unterdrückt.

Da es Parallelrechnersysteme gibt, die eine spezielle Endebehandlung mit Freigabe von Ressourcen verlangen (z. B. PVM), wurde ein zusätzliches Unterprogramm

```
SUBROUTINE TRCLOSE
```

zum Abschließen des Hauptprogramms vorgesehen, das stets am Ende eines Programms (auf jedem Prozessor) gerufen werden sollte (anstelle einer `STOP`-Anweisung).

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'` bzw. `#include "trnet.h"`

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)<br>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. (i. a. gilt: `ICH` $\neq$ `ICHRING`, lediglich für `ICH` $= 0$ gilt stets auch: `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 *Broadcast* von Prozessor 0 zu allen anderen Prozessoren; es existieren spezielle Varianten dieser Routine: |
| `TREE_DOWN_0` | Die Länge ist im voraus bekannt |
| `MTREE_DOWN` | Individuelle Daten für jeden Prozessor |
| `MTREE_DOWN_0` | Individuelle Daten, aber konstante Länge für jeden Prozessor |
| `TREE_UP` | Schnelles *Zusammenfassen* der Daten von allen Prozessoren zum Prozessor 0 (Speicherkapazität beachten!) |
| `TREE_UP_0` | Variante von `TREE_UP`, gleiche Länge von allen Prozessoren |
| `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 |
| `V_CUBE_EXCH` | Nach Prozessornummern geordnetes *Einsammeln* von Daten ungleicher Länge von allen Prozessoren zu allen Prozessoren |
| `TOT_EXCH` | Eine *Total-Exchange*-Routine, bei der jeder Prozessor jedem anderen ein individuelles Paket gleicher Länge sendet (im Gegensatz zu `CUBE_EXCH`, wo er jedem Prozessor das gleiche Paket schickt) |
| `TOT_CAT` | Eine weitere *Total-Exchange*-Routine, mit individuellen Datenpaketen unterschiedlicher Länge von jedem Prozessor für jeden anderen |
| `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_IN` | Zusammen mit `RING_SEND0` (als Gegenstück zu `RING_OUT` und `RING_RECEIVE0`) können Daten von Prozessor 0 eingelesen und paketweise durch den Prozessorring verteilt werden. |
| `RING_FORW` | Im eingebetteten Prozessor-Ring sendet jeder Prozessor Daten an den Nachfolger und empfängt Daten von seinem Vorgänger |
| `RING_BACK` | Im eingebetteten Prozessor-Ring sendet jeder Prozessor Daten an den Vorgänger und empfängt Daten von seinem Nachfolger |

Bei allen genannten Operationen wird vorausgesetzt, dass sie bezüglich des logischen Programmablaufs auf allen Prozessoren „**gleichzeitig**“ gerufen werden. Die einzige Ausnahme (die eigentlich auch keine ist) gibt es bei den Unterprogrammen `RING_OUT` und `RING_IN`, die nur auf den Prozessoren `ICH`> 0 gerufen werden, während Prozessor 0 eine andere Routine `(NPROC-1)`-mal ruft (`RING_RECEIVE0` bzw. `RING_SEND0`).

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 (3.3).

Am Ende dieses Abschnitts folgt eine kurze beispielhafte Erläuterung zur Wirkungsweise der Kommunikationsroutinen (3.4).

## 3.2 Beschreibung der Kommunikationsroutinen

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

**Rufzeile:**

```
call  TREE_DOWN    ( N, X )
call  TREE_DOWN_0  ( N, X )
call  MTREE_DOWN   ( N, X )
call  MTREE_DOWN_0 ( N, X )
```

**Parameter:**

- `N` - Länge des zu sendenden (bzw. empfangenen oder zu empfangenden) Datenfeldes; gezählt in Worten
- `X` - Vektor mit `N` Worten (oder $\frac{N}{2}$ Doppelworten), der von Prozessor 0 gesendet und von allen anderen Prozessoren empfangen wird; bzw.
  - auf Prozessor 0 der Vektor, der die verschiedenen Datenpakete für die anderen Prozessoren enthält; auf jedem anderen Prozessor der Vektor, der zum Schluss die für ihn bestimmten Daten enthält (eine ausreichende Länge des Feldes `X`, im Falle von `MTREE_DOWN` und `MTREE_DOWN_0` auch für das Zwischenspeichern von Daten für andere Prozessoren, wird vorausgesetzt)

**Funktion:** Daten, die Prozessor 0 im Dialog mit dem Nutzer am Hostrechner (oder aus Eingabefiles) erhalten hat, sollen allen Prozessoren zugänglich gemacht werden.

Dies geschieht hier durch das baumartige Versenden im Hypercube (in `NCUBE` Zeitschritten).

Die speziellen Varianten berücksichtigen bestimmte Sonderfälle:

`TREE_DOWN_0` Hier wird vorausgesetzt, dass die Länge des *einen* Datenpaketes, das von Prozessor 0 an alle anderen Prozessoren geschickt wird, bereits auf allen Prozessoren bekannt ist. So kann die *Anzahl* der Kommunikationen auf die Hälfte reduziert werden.

`MTREE_DOWN_0` Prozessor 0 besitzt für jeden anderen Prozessor ein spezifisches Datenpaket; das Feld `X` enthält daher auf Prozessor 0 `N*NPROC` Worte, von denen jeder Prozessor *seine* `N` Worte erhält.

`MTREE_DOWN` Die verschiedenen Datenpakete für die einzelnen Prozessoren können hier auch unterschiedliche Längen haben. Deshalb muss der erste Parameter `N` ein **`Integer-Feld (!)`** der Länge `NPROC` sein, das auf Prozessor 0 die einzelnen Paketlängen enthält – auf den anderen Prozessoren wird es zunächst als Hilfsfeld benutzt und liefert am Ende in der ersten Komponente die Länge der empfangenen Daten.

**Bemerkungen: Zu beachten** ist, dass in jedem Fall `X` und im Fall von `TREE_DOWN` und `MTREE_DOWN` auch der Parameter `N` für alle Prozessoren `ICH` $> 0$ Output-Parameter sind, d. h. insbesondere, dass `N` nie als Konstante angegeben werden sollte, da diese sonst durch den von Prozessor 0 gesendeten aktuellen Wert überschrieben würde. Das kann (z.B. bei Linux) dazu führen, dass die betreffenden Programme wegen Speicherschutzverletzung abbrechen.

Die Varianten `MTREE_DOWN` und `MTREE_DOWN_0` setzen voraus, dass hinreichend Speicherplatz vorhanden ist, um alle Daten zugleich auf einem Prozessor unterzubringen, bevor sie verteilt werden. Alternativ steht die Kombination der Programme `RING_SEND0` und `RING_IN` zur Verfügung (3.2.11), wo jeweils nur Platz für **ein** Datenpaket erforderlich ist, dabei aber die Kommunikation mehr Zeit benötigt.

Noch ein **Tip**: Mit `N=0` dient das Programm `TREE_DOWN` 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. Zusammen mit einem vorhergehenden `TREE_UP` mit Länge 0 ist außerdem gesichert, dass Prozessor 0 das *Signal* erst dann gibt, wenn alle Prozessoren dieses erwarten. Das alles geht natürlich auch mit `N>0`. Dagegen ist aber `Tree_Down_0(0,X)` nicht für derartige Zwecke geeignet, weil keine Kommunikation und damit keine Synchronisation stattfindet.

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

```
      PROGRAM  DEMO_0815
       ...
      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 'include/trnet.inc'

      call TRINIT ( 0 )
C
C Erst jetzt kennt jeder Prozessor seine Identitaet !
C Um ein Bildschirm-Chaos zu vermeiden, soll nur Prozessor 0
C den Dialog mit dem Nutzer fuehren.
C
      IF ( ICH .EQ. 0 ) THEN
    1    write (*,'(A,$)')   ' Dimension, Epsilon = '
         read (*,*,ERR=1) N, EPS
         IF ( N .GT. 0 ) THEN
           write (*,'(A,$)') ' Eingabedatei: '
           read (*,'(A)') FNAME
           call FRCHAN (log_unit)
           OPEN  (UNIT=log_unit, FILE=FNAME, FORM=UNFORMATTED)
           READ (log_unit) (HELP(I),I=1,N)
           CLOSE (UNIT=log_unit)
         END IF
      END IF
C
C Der Wert L_INFO ist auf allen Prozessoren schon bekannt, deshalb
C TREE_DOWN_0 statt TREE_DOWN moeglich:
C
      call TREE_DOWN_0 (L_INFO,INFO)
C
C Variablenwerte N und EPS nun auf allen Prozessoren bekannt
C
      IF ( N .GT. 0 ) THEN
        L=2*N         ! doppelte Anzahl Worte, (DOUBLE PRECISION)
        call TREE_DOWN_0 (L,HELP)
         ...
      ELSE
        call TRCLOSE
      ENDIF
      END
```

### 3.2.2 TREE_UP - Zusammenfassung von Daten aller Prozessoren

**Rufzeile:** `call TREE_UP   ( Nlocal, Xin, Nout, Xout, MaxX )`
`call TREE_UP_0 ( 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` [6]

**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.

Prozessor 0 erfährt bei `TREE_UP` nur die Gesamtlänge `Nout` der empfangenen Daten, nicht die einzelnen Längen von jedem Prozessor. Sollte diese Information gebraucht werden, muss man sich z. B. zusätzlich ein

```
call TREE_UP_0 (1,Nlocal,Nout,NLocals,NPROC)
```

leisten, mit einem `Integer`-Feld `NLocals(NPROC)`.

Die Variante `TREE_UP_0` kann vorteilhaft angewendet werden (weniger Kommunikationen), wenn bekannt ist, dass `Nlocal` auf allen Prozessoren gleich ist. Obwohl der Parameter `Nout` dann immer den Wert `Nlocal*NPROC` hat, wurde er wegen der Analogie zu `TREE_UP` in der Rufzeile beibehalten.

**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 'include/trnet.inc'
       ...
      NH(1)=NC
      NH(2)=NI
      call TREE_UP_0 ( 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
       ...
```

[6] Die Größe `MaxX` wird vorläufig aus Effektivitätsgründen nicht berücksichtigt.

### 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 (vgl. Beispiel). Der Vorteil dieser Routine liegt in der Möglichkeit, auf jedem Prozessor unterschiedlich lange Datenfelder bereitzustellen (auch Länge `Nlocal=0`). Durch den Verzicht auf eine geordnete Reihenfolge werden Kopieroperationen im Speicher vermieden.

Für den Austausch unterschiedlich langer Datenpakete in geordneter Reihenfolge steht das Unterprogramm `V_CUBE_EXCH` zur Verfügung (3.2.5).

**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 einem speziellen Programm VECADI die lokal interessierenden
C Daten C entsprechend der Indizes Ihelp(*) aus dem (globalen)
C Vektor HELP selektiv akkumulieren :
       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).

**Bemerkungen:** Die Ordnung mittels `ID=ICHRING` kann nur bei der „klassischen" Einbettung des Prozessorrings in den Hypercube verwendet werden (vgl. 3.4). Unter Parix wird durch `TRINIT` allerdings eine für die Hardware besser geeignete Anordnung der Prozessoren im Ring gewählt. Die klassische Variante lässt sich aber nach `TRINIT` durch den zusätzlichen Aufruf `call CUBE_DIM(NCUBE)` erzwingen.

Die Felder `Xin` und `Xout` dürfen identisch sein.

Die Routine `CUBE_EXCH` kann nur dann verwendet werden, wenn `Nlocal` auf allen Prozessoren gleich ist (vgl. auch 3.2.5 und 3.2.6).

**Beispiel:** Die zeitaufwendige Berechnung eines Vektors, den alle Prozessoren benötigen, soll lokal ausgeführt werden, um anschließend die Teilergebnisse wieder an alle Prozessoren zu verteilen:

```
        ...
       include '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 `V_CUBE_EXCH` - Austausch von Datenpaketen verschiedener Länge

**Rufzeile:** `call V_CUBE_EXCH ( Nlocal, Xin, Xout, Nptrs, 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 $\sum_p (\mathtt{Nlocal}_p)$
- `Nptrs` - Hilfsfeld (`Integer Nptrs(0:NPROC)`), auf dem die Startindizes der einzelnen Teilfelder der Prozessoren innerhalb von `Xout` zurückgegeben werden; Länge ist `NPROC+1`
- `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).

**Bemerkungen:** Es treffen dieselben Anmerkungen zu wie zuvor bei `CUBE_EXCH` (3.2.4), außer der Forderung nach gleichen Paketlängen.

**Beispiel:** Der lokale Anteil eines verteilt zu berechnenden Vektors ist auf den Prozessoren unterschiedlich. Alle Prozessoren benötigen anschließend den gesamten (globalen) Vektor (z.B. um zu häufige Kommunikationen zu vermeiden):

```
       ...
      include 'include/trnet.inc'  ! Hypercube-Parameter
      DOUBLE PRECISION X(*)        ! Laenge:  NPROC*Nlocal
      INTEGER nPtrs(0:NPROC)       ! Startindizes
       ...
C Nlocal = f(ICH)
C Berechnung der Komponenten X(1), ..., X(Nlocal)
C mit X(i) = f ( Nlocal,i )
       ...
      call V_CUBE_EXCH ( Nlocal, X, X, nPtrs, ICH )
C Der Vektor X steht nun in voller Laenge auf allen Prozessoren.
C mittels X(nPtrs(k)) kann man den von Prozessor k (0,...,NPROC-1)
C berechneten Teilvektor adressieren.
C nPtrs(NPROC)-1 ist die Gesamtlaenge des Vektors X
       ...
```

### 3.2.6 TOT_EXCH/TOT_CAT - Total-Exchange-Varianten

**Rufzeile:** `call TOT_EXCH ( Nlocal, X, H )`
`call TOT_CAT  ( Nlocal, MaxLen, X, H )`

**Parameter:**

- `Nlocal` - Länge der einzelnen Felder `X(*)`, die der Prozessor für jeden anderen liefert; gezählt in Worten
  - im Fall `TOT_CAT`: ein Vektor mit `NPROC` Einträgen für die Länge der einzelnen Pakete, die den anderen Prozessoren zu senden sind.
- `MaxLen` - (auf allen Prozessoren vor der Operation bekannte!) maximale Länge eines Datenpakets (ggf. mit `CUBE_DOI(...,VImax)` ermitteln);
- `X` - Datenfeld, das der aktuelle Prozessor (komplett oder teilweise) an alle anderen zu schicken hat und auf dem die Daten der anderen Prozessoren empfangen werden sollen; Länge `Nlocal*NPROC` bzw. `MaxLen*NPROC`
- `H` - Hilfsfeld zur Realisierung der Kommunikation; Länge `Nlocal*NPROC/2` bzw. `MaxLen*NPROC/2`

**Funktion:** Es erfolgt ein globaler Austausch von Daten, wobei jeder Prozessor für jeden anderen Prozessor je ein individuelles Paket besitzt, diese Pakete stehen lückenlos hintereinander auf `X`. Die Länge der zu versendenden Daten ist bei `TOT_EXCH` für alle Pakete von allen und für alle Prozessoren konstant und bei `TOT_CAT` völlig beliebig, also bis zu $[\texttt{NPROC}]^2$ verschiedene Längen. Durch einen Parameter `MaxLen`$\leq 0$ wird das Programm `TOT_CAT` dazu veranlasst, das Maximum durch zusätzliche Kommunikation zunächst selbst zu bestimmen. Am Ende der Operation hat jeder Prozessor auf `X` die aneinandergereihten, für ihn bestimmten Datenpakete von den anderen Prozessoren, wieder in geordneter Reihenfolge.

**Beispiel:** Die Routine kann auch benutzt werden, um etwa eine vom Hypercube abweichende Nachbarschaft von Prozessoren zu realisieren:

```
         ...
        DOUBLE PRECISION X(*), H(*)
        INTEGER          INFO(3,L_Info), Nloc(MAX_PROC)
         ...
        call VICOPY (NPROC, Nloc, 1, 0, 0)   ! alle Laengen := 0
        jh = 1
        DO i=1,L_Info
C   Annahme: Jede Prozessornummer hoechstens einmal in INFO
C            und schon nach Prozessornummer geordnet.
          ip=INFO(1,i)        ! Nachbarprozessor
          ia=INFO(2,i)        ! Start-Index in X
          ie=INFO(3,i)        ! Ende-Index in X
          Nloc(ip) = ie-ia+1  ! Laenge fuer Proz. ip
          call VIcopy (Nloc(ip), H(jh),1, X(ia),1)
          jh=jh+Nloc(ip)
        ENDDO
        MaxLen=0
        call VImax (NPROC, MaxLen,0, Nloc,1)            ! lokales Max.
        call CUBE_DOI (1, MaxLen, MaxLen, H(jh), VImax) ! globales Max.
        jh=NPROC*MaxLen+1
        call TOT_CAT ( Nloc, MaxLen, H, H(jh) )  ! Ergebnis auf (Nloc,H)
         ...
```

### 3.2.7 `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`.

Man **beachte**, dass `VtOP` im rufenden Programm als `EXTERNAL` vereinbart sein muss!

**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
- `AND` - bitweise logisches **AND**
- `OR` - bitweise logisches **OR**
- `EOR` - bitweise logisches exklusives **OR** (**XOR**)

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)
 ...
```

### 3.2.8 `TREEUP_DOD/TREEUP_DOI` - Operationen über alle Prozessoren

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

**Parameter:**

- `N` - Länge der Operandenfelder je Prozessor
- `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:** Ähnlich wie bei `CUBE_DOx` werden mittels einer kommutativen Operation `VtOP` die Daten von allen Prozessoren miteinander verknüpft.

**Im Gegensatz zur Operation `CUBE_DOx` liegt das Ergebnis nach dieser Operation nur auf Prozessor 0 vor.**

Ansonsten ist die Bedeutung der Parameter analog. Die vom aktuellen Prozessor zu liefernden Daten befinden sich im Vektor `Y`, dessen Typ zum Typ `"t"` der auszuführenden Operation `VtOP` passt (vgl. 3.2.7).

Man **beachte**, dass `VtOP` im rufenden Programm als `EXTERNAL` vereinbart sein muss!

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:** Prinzip einer 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, so kann auf den
C          anderen Prozessoren KettAkk (mit Kommunikation)
C          zeitgleich mit SOLVER auf Prozessor 0 laufen,
C          wenigstens teilweise.
C
      NW=2*NCrossG*Nfg
      call TREE_DOWN_0 (NW,W)
       ...
```

### 3.2.9 `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 'include/trnet.inc'
         ...
        call TrInit(0)
        NCUBE_orig = NCUBE
1       CONTINUE
        IF (ICH .EQ. 0) THEN
          write (*,'(A,$)') ' Cube-Dimension: '
          read  (*,*,ERR=1)   NewDim
          IF ( NewDim .GT. NCUBE_orig ) GOTO 1
        ENDIF
        L=1
        call TREE_DOWN_0 (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
         ...
```

### 3.2.10 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 Grafik-Koordinaten, die jeder Prozessor für sein Teilgebiet berechnet hat:

```
         ...
        DIMENSION  X(3,*), HELP(*)
        include '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
```

### 3.2.11 `RING_IN` - Eingabe über Prozessor-Ring

**Rufzeile:** `call RING_IN ( Nlocal, X, MaxX )`

**Parameter:**

- `Nlocal` - Länge des vom aktuellen Prozessor empfangenen Datenfeldes (in Worten)
- `X` - Datenfeld, das zunächst als Hilfsfeld zum „Durchreichen“ an die Ringnachfolger benutzt wird und am Ende die für den aktuellen Prozessor bestimmten Daten enthält
- `MaxX` - verfügbare Länge des Feldes `X`; muss mindestens Maximum aller Werte `Nlocal` (für alle Prozessoren) sein

**Funktion:** Diese Funktion sollte anstelle von `TREE_DOWN` bzw. `MTREE_DOWN` verwendet werden, um unterschiedliche Datenpakete von Prozessor 0 an alle anderen Prozessoren zu verteilen, falls diese nicht zugleich im Speicher eines Prozessors unterzubringen sind; z. B. wenn längere Datenfelder, die mittels `RING_OUT` nach stundenlanger Rechnung auf ein File gerettet wurden, zwecks Wiederanlauf wieder eingelesen und verteilt werden sollen.

Auch diese Routine arbeitet im wesentlichen sequentiell im Prozessor-Ring. Sie darf nicht auf Prozessor 0 verwendet werden, dort muss statt dessen insgesamt `(NPROC-1)`-mal die Routine `RING_SEND0` aufgerufen werden (siehe Beispiel!).

`RING_IN` transportiert in Richtung von Link `Lforw` (genau wie `RING_OUT`), so dass die Daten automatisch genau so verteilt werden, wie sie es vor `RING_OUT` waren. Eine Sonderbehandlung ist eventuell für die Daten von Prozessor 0 selbst erforderlich, wenn diese als erste ausgegeben wurden, aber als letzte wieder eingelesen werden sollen.

**Beispiel:** Einlesen von Daten, die zuvor mittels `RING_OUT` in ein File geschrieben wurden.

```
       ...
      DIMENSION  X(*), H(*)
      include 'include/trnet.inc'
       ...
      IF ( ICH .GT. 0 ) THEN
        call RING_IN ( Nlocal, X, MaxX )
      ELSE
C       auf Prozessor 0: zuerst eigene Daten lesen, die bei
C       RING_OUT zuerst ausgegeben wurden:
        call Einlesen ( Nlocal, X, ... )
C       dann die von allen anderen Prozessoren verteilen
C       (falls wegen Speicherplatzmangel kein zusaetzliches
C        Hilfsfeld H vorhanden ist, muss hier das Feld X
C        verwendet werden, und die eigenen Daten sind am Ende
C        nochmals einzulesen)
        DO  I=2,NPROC
          call Einlesen ( N, H, ... )
          call RING_SEND0 ( N, H, MaxX )
        END DO
C       und zuletzt nochmal die eigenen Daten lesen
C evtl.       REWIND  log_unit
C evtl.       call Einlesen ( Nlocal, X, ... )
      END IF
```

### 3.2.12 `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
       ...
```

## 3.3 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[7] 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, oder die durch Multitasking bedingten Wartezeiten auf einer Workstation. Die Zeit für ein Programm wird z. B. nach folgendem Prinzip gemessen:

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

Im rufenden Programm muss `SECNDS` manchmal als `EXTERNAL` vereinbart sein (in der Regel bei GNU-Fortran-Compilern), um eine gleichnamige Intrinsic-Function zu ersetzen, die nur ganzzahlige Sekundenwerte liefert. 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 interne Messung zu nutzen, die zugleich auch die Auswertung und Ausgabe der entsprechenden Werte von allen Prozessoren einschließt, können wir folgende Routinen empfehlen, die mit Ausnahme von `WRITE_TIMES` auf jedem Prozessor gerufen werden müssen:

| | |
|---|---|
| `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 (`H`: 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; |
| `TIME_PS(H)` | - dasselbe als Postscript-File; diese beiden Routinen befinden sich in der Bibliothek `libGraf.a` ! |

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

- Prozessornummer `ICH`
- unter Parix die $x, y, z$-Koordinaten des Prozessors im Parix-Gitter
- Kommunikationszeiten für `RECV`[8] seit letztem `Init_Time` in Sekunden und in Prozent der gesamten Laufzeit
- Kommunikationszeiten für `SEND`[8] (in analoger Weise)
- Gesamtlaufzeit, ebenfalls gemessen seit letztem `Init_Time`

Für große Prozessoranzahlen wurde bei der Ausgabe der Tabelle (zwecks Überschaubarkeit am Bildschirm) eine Beschränkung auf maximal 16 Einträge vorgenommen.

---

[7]Diese Funktion `SECNDS` ist nur teilweise bereits Compiler-Standard, z. B. durch spezielle Flags, unter HP-UX mit “`+E1`” oder unter SunOS mit “`-lV77`”. Für andere Systeme wurde ein entsprechendes Funktionsunterprogramm in die Bibliothek `Cubecom` aufgenommen.

[8]Die Kommunikationszeiten für `SEND` bzw. `RECV` werden innerhalb der Routinen `SEND_CHAN_0` und `SEND_NODE_1` bzw. `RECV_CHAN_0` und `RECV_NODE_1` gemessen und aufsummiert, ohne die verschiedenen Kommunikationsvarianten zu unterscheiden.

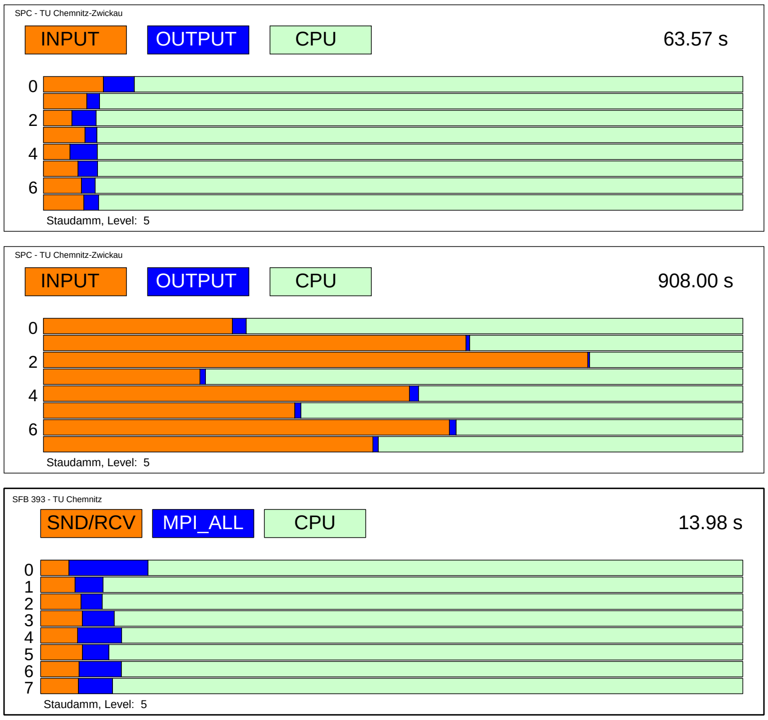


Abbildung 1: Ausgabe von `Time_PS`, z. B. für GCPowerPlus (oben) bzw. PVM auf einem Sun-Workstation-Cluster und MPI auf dem Linux-Cluster CLiC (Lösung eines Gleichungssystems mit 522432 Unbekannten),
Bem.: Für vergleichende Darstellungen ist eine Skalierung der Balkenlänge im Diagramm möglich (siehe Kommentare im Postscriptfile).

Beim Aufruf von `TIME_GRAF` wird in einem separaten Grafikfenster 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 (siehe Abb. 1).

Beim Aufruf von `TIME_PS` wird ein solches Diagramm als Postscript-File (EPS-Format) ausgegeben und `TIME_GRAF` zur Ausgabe auf den Bildschirm ebenfalls mit ausgeführt.

Die als Differenz von Gesamtlaufzeit und gemessener Kommunikationszeit ausgewiesene CPU-Zeit ist als solche auf „echten“ Parallelrechnern sehr zuverlässig.

Auf Workstation-Clustern (PVM) ist die Bezeichnung „CPU-Zeit“ zwar nicht gerechtfertigt, der Wert ist aber trotzdem aussagekräftig. Eine Maschine mit hoher „CPU-Zeit“ ist entweder langsamer oder durch Multitasking zeitweise stärker belastet als andere. Dadurch entstehen wiederum auf den schnelleren Maschinen relativ hohe Kommunikationszeiten. Die Darstellung der Kommunikationszeit erfolgt entweder durch die beiden Anteile `INPUT` und `OUTPUT` oder bei Verwendung der speziellen MPI-Variante der Bibliothek `Cubecom` durch die beiden Anteile `SND/RCV` für die elementaren `send`- und `recv`-Operationen und `MPI_ALL` für alle kollektiven Kommunikationen der Art `MPI_Allreduce, MPI_Allgather`. Diese Operationen können auch Arithmetikzeit einschließen, die bei anderen Bibliotheksversionen zur CPU-Zeit gerechnet wird.

## 3.4 Zur Wirkungsweise der Kommunikationsroutinen

Alle im Abschnitt 3.2 genannten Kommunikationsroutinen beruhen auf der (virtuellen) Hypercube-Topologie des Parallelrechners und den damit verbundenen einfach beherrschbaren Nachbarschaftsverhältnissen der Prozessoren (siehe [8, 9]).

**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 $2^{k-1}$ (vgl. Abb. 2). Daher kann Prozessor `ICH` die Nummer jedes seiner Nachbarn anhand der Link-Nummern bestimmen:

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

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 Links mit niedrigerer Nummer weiterzugeben. Diese erste Link-Nummer entspricht der Position des letzten Nicht-Null-Bits in `ICH`. So erhalten beispielsweise alle Prozessoren mit ungerader Nummer ihre Daten erst im letzten Schritt über Link 0 (vgl. Abb. 3).

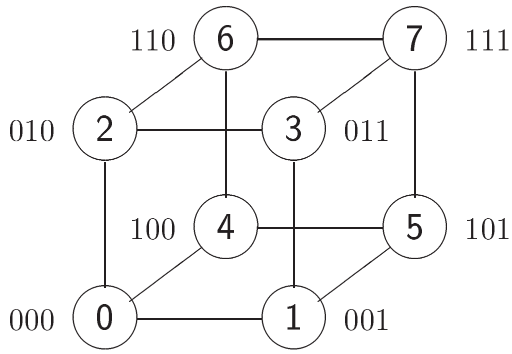


Abbildung 2: Hypercube-Grundstruktur

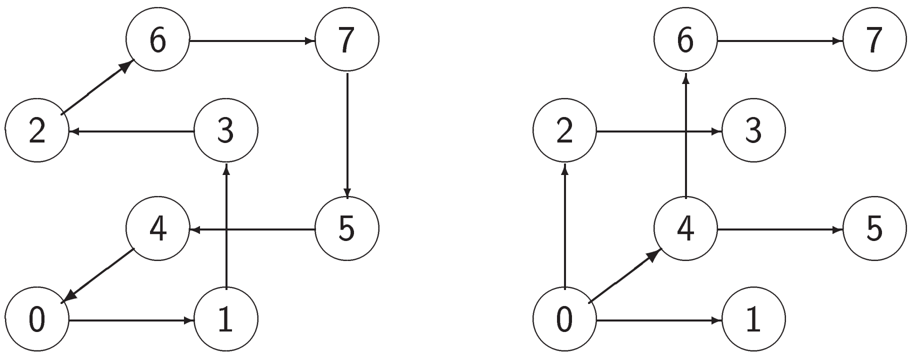


Abbildung 3: Hypercube-Substrukturen: Ring und Baum

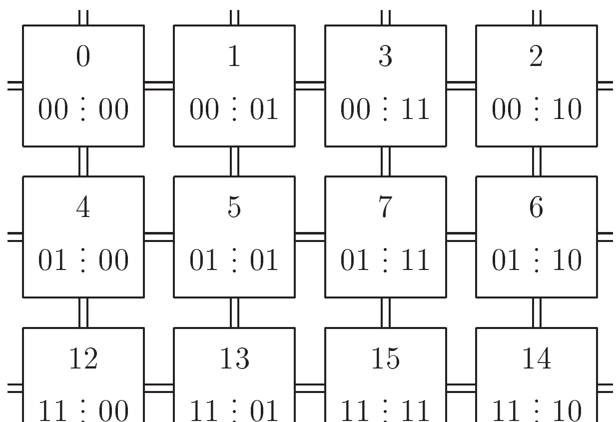


Abbildung 4: Hypercube-Substruktur: 2D-Gitter/Torus

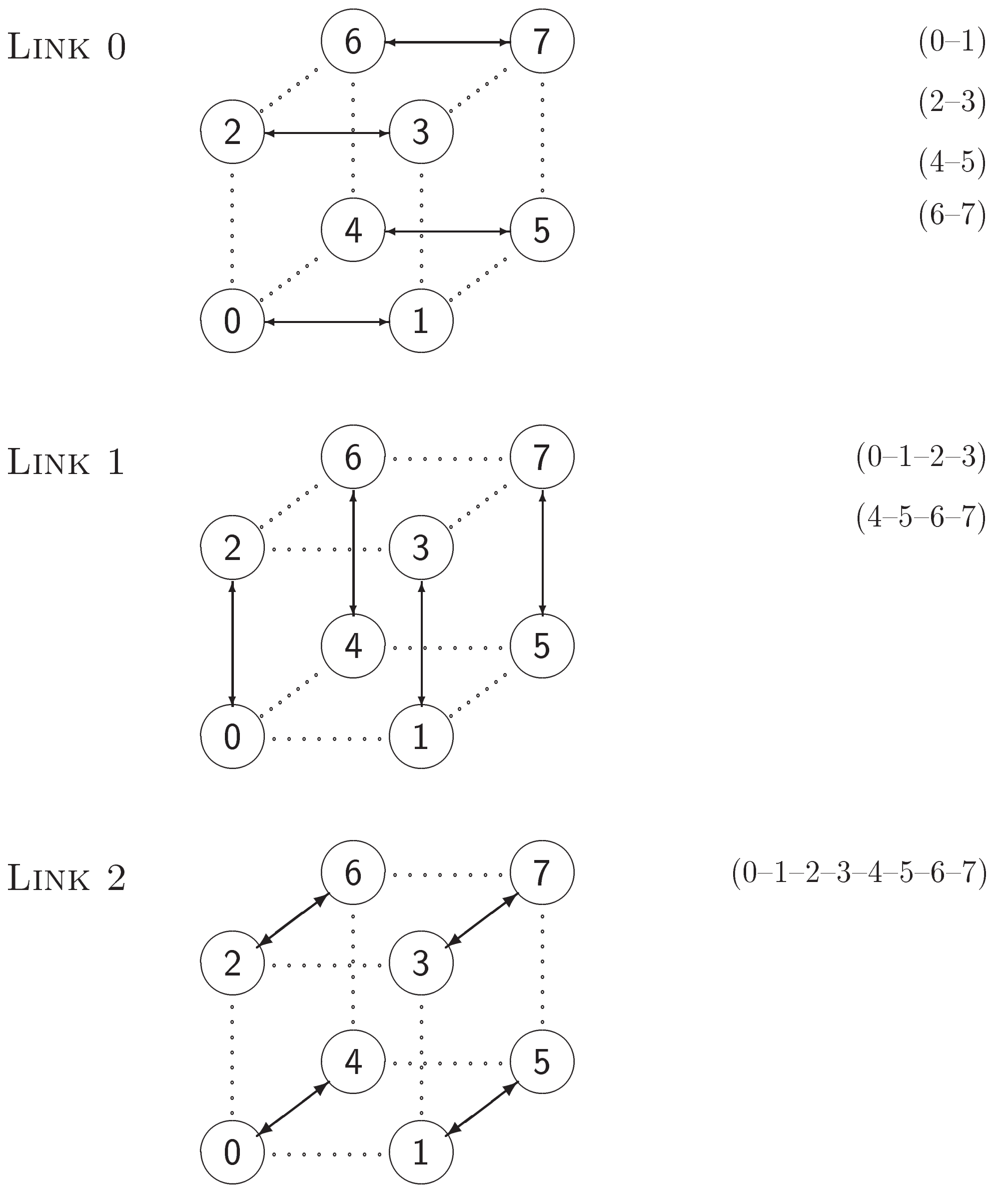


Abbildung 5: Datenfluss bei `Cube_Do`

Eine der wichtigsten Kommunikationsroutinen ist die Operation `CUBE_DOx` zur Realisierung globaler Operationen über alle Prozessoren (Abschnitt 3.2.7). 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, vgl. Abb. 5).

```
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 bereits 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[9] war es, einfache Grafikausgaben (passiv) auf UNIX–Workstations unter FORTRAN zu realisieren, wobei der Anwendungsprogrammierer nicht die Vielfalt der `Xlib`–Bibliotheksaufrufe beherrschen muss und insbesondere auch nicht das ereignisgesteuerte Programmiermodell in seinem Programm berücksichtigen muss, wie es sonst bei X–Anwendungen üblich ist ([2], [5], [7]). Die Schnittstelle ist gleichermaßen für normale Workstation-Anwendungsprogramme wie auch für parallele Anwendungen unter Parix oder PVM nutzbar.

Es können mehrere Fenster zur Grafikdarstellung geöffnet werden. Obwohl der Schwerpunkt auf passiver Grafik liegt, werden auch Routinen zur Eingabe von Zahlen und Zeichenketten oder zur Tastatur– und Mausabfrage bereitgestellt.

Die hier vorgestellten Routinen bilden den Kern der Bibliothek `libGraf.a` zur Darstellung von Grafik unter X–Windows. Darauf aufbauend, enthält diese Bibliothek weitere, problemspezifische Unterprogramme (vgl. [4], [6]).

## 4.2 Bemerkungen zur Funktionsweise

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

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. Die Angabe `-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.

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 X–Terminals mit relativ wenig Speicher kann es vorkommen, dass der Fensterinhalt nach Überlappung mit einem anderen Fenster nicht automatisch wiederhergestellt werden kann. Das hat allerdings nichts mit dem Speicherplatz des (Parallel-)Rechners zu tun, auf dem das Programm selbst läuft.

---

[9] Die ursprüngliche Version des Programms (mit separatem Server-Programm `gserver`) und die Version mit direkten X11-Funktionen wurden von Thomas Hommel entwickelt. Einige Modifikationen stammen von Matthias Pester und Michael Seibt.

## 4.3 Beschreibung der Unterprogramme der X11-Schnittstelle

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. Da in Fortran Zeichenkettenvariablen immer bis zur vereinbarten Länge mit Leerzeichen aufgefüllt werden, kann es manchmal günstiger sein, eine Teilzeichenkette minimaler Länge an das Grafikprogramm zu übergeben. Mit dem Hilfsprogramm (`Len_Trim`), aus der Bibliothek `libTools.a` kann man das leicht realisieren:

```
      CHARACTER*(*) Filename
      CHARACTER*80  Zeile
        ...
      write(Zeile,100) Filename,N
100   Format ('Beispiel: ',A,' - Dimension:',I8)
      L=Len_Trim(Zeile)        ! "echte" Laenge ermitteln
      call gstorename ( iwin, Zeile(1:L) )
```

Die Namensgebung der UP–Rufe ist an die Namen der entsprechenden X–Rufe angelehnt.

### 4.3.1 Initialisierungs- und Window-Operationen

`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 GSetDisplay ( displayname )`
Dieses Programm kann **vor** dem Programm `GOpenServer` aufgerufen werden, damit dieses anstelle der Environmentvariablen `DISPLAY` das angegebene Display verwendet. Der Aufruf ist auch noch möglich, nachdem `GOpenServer` erfolglos gerufen wurde und ein neuer Versuch gestartet werden soll.
`displayname` ist eine Zeichenkette der Form `"hostname:0"`.

`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 manuell 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. Größe und Position werden in Form einer

Zeichenkette angegeben wie bei X-Anwendungen im `-geometry`-Parameter üblich: `"<breite>x<hoehe>±<xpos>±<ypos>"`. – Es müssen stets alle 4 Werte angegeben werden (was bei vielen anderen Anwendungen nicht notwendig ist).

`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 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 GGetSizes ( iwin, ix, iy, jx, jy, iscrnx, iscrny, mmx, mmy )`
Es werden mehrere Größenangaben geliefert: `(ix,iy)` entspricht den Ausmaßen des Fensters wie bei `GGetSize`. Die Werte `(jx,jy)` geben die Position des Fensters auf dem Bildschirm an (linke obere Ecke, allerdings nicht immer zuverlässig nach manueller Größenänderung). Die Werte `(iscrnx,iscrny)` und `(mmx,mmy)` geben die Größe des gesamten Bildschirms an, entsprechend in Pixeln und in „Millimetern“ (letzteres ist auch nicht zuverlässig).

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

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

`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 GBell`
Ein akustisches Signal wird ausgegeben.

### 4.3.2 Farben, Pixmap

`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. Am häufigsten dürfte „PseudoColor" vorkommen, wobei jeder Farbe ein Index (0. . . 255) zugeordnet wird, dessen konkrete Farbwerte (RGB) separat vom X-Server verwaltet werden (Palette).

`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 $0 \ldots 2^{16} - 1$). Zurückgegeben wird die Kennzahl `icolor`, die dann z.B. bei `GSetForeGround` benutzt werden kann. Man beachte aber, dass die Anzahl der Farbeinträge je nach Display beschränkt ist und deshalb u. U. nicht die gewünschte Farbe eingetragen werden kann (vgl. `GSwitchColorMap`).

`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 GParseColInd ( ir, ig, ib, icolor)`
Für die angegebene Farbkennzahl `icolor`, die aus einem vorhergehenden Zuweisung (`GAllocColor, GRainColor, ...`) stammt, werden die Rot–Grün–Blau–Anteile in `ir, ig, ib` zurückgegeben.

`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 GSwitchColorMap`
Es wird zwischen der Standardfarbpalette und einer *privaten* Farbpalette des laufenden Programms umgeschaltet. Dies kann erforderlich werden, weil (je nach Hardware) eine Farbpalette z. B. nur 256 verschiedene Farben enthält. Wenn mehrere farbintensive Anwendungen auf einem X-Terminal laufen, ist diese Palette schnell erschöpft und `GAllocColor` kann keine neuen Farbindizes mehr liefern (erkennbar durch `icolor`=0). Durch das Umschalten auf eine private Palette hat das Programm wieder eine Palette mit bis zu 256 Einträgen für sich selbst zur Verfügung. Auf dem Bildschirm führt das zu einer „Falschfarbendarstellung" in den Bereichen der jeweils nicht aktiven Farbpalette. Bei wiederholtem Aufruf schaltet das Programm wieder auf die Standardpalette zurück (usw.); nach jedem Umschalten sind alle Farben wieder neu zu definieren (`GAllocColor`). Es ist dringend zu empfehlen, auch vor dem Umschalten der Palette alle zuvor definierten Farben freizugeben (`GFreeColor`).

`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.

### 4.3.3 Zeichenoperationen

`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. Das Feld `points` hat also mindestens die Länge `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**, `mode = 6` → **xor** oder `mode = 1` → **and**.
Im Normalfall wird direkt gezeichnet (**copy**). Die entsprechende logische Verküpfung 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 (in Abhängigkeit von den vorherigen Pixelfarben im Fenster) zu sehen ist.

### 4.3.4 Text im Grafikfenster

`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 unterscheidet. 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 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.

### 4.3.5 Eingaben über Tastatur und Maus

`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 (als `Integer`-Wert 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 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 gleichzeitig 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 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 oder 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 (kleiner `'␣'`) sein 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 durch die Unterprogramme selbst 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 Grafikbibliothek 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 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 3.2.10). 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 sind einige der Unterprogramme in einer (durch Compilerflag) modifizierten Variante neu zu übersetzen. 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`).

Intern werden Kommunikationsroutinen der Bibliothek `Cubecom` verwendet. Es sei noch einmal erwähnt, dass diese Mehrprozessorvariante für große Prozessoranzahlen ($\geq 32$) nicht effektiv ist bzw. aufgrund interner Beschränkungen von X11 nicht mehr möglich ist. Diese Version liegt wird deshalb auch nicht als Bibliothek vor.

## 4.6 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 Display 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, Programm beenden  ----------------------
        call  GClosewin     ( iwin1 )
C
        end
```

## 4.7 Einige Zusatzfunktionen

Die hier aufgeführten Unterprogramme arbeiten nach dem Client-Server-Prinzip mit einem Programm auf der lokalen Maschine des Nutzers zusammen, indem dieses mit Daten zur weiteren Verarbeitung versorgt wird.

### XXgrab - Image abspeichern

Dieses Programm kopiert den Fensterinhalt als Bitmap und speichert ihn in einer Datei im GIF-Format. Nachdem das Programm auf der Workstation gestartet wurde, wartet es auf *Kommandos*, die (vom Parallelrechner) durch die folgenden Unterprogramme geliefert werden:

`call GrabInit ( iwin, ierr )`
Initialisierung von `XXgrab`. Es wird mitgeteilt, mit welchem Fenster gearbeitet wird. Dabei wird auch die Farbpalette mit übernommen (sie sollte also nach `GrabInit`) nicht gewechselt werden.
`GrabInit` versucht zunächst, das Programm `XXgrab` auf dem Rechner zu „finden“, der über die `DISPLAY`-Variable definiert ist. Kommt keine Verbindung zustande, wird nach dem Hostnamen gefragt, auf dem dieses Programm läuft. Prinzipiell ist damit auch die folgende Situation möglich: Der Nutzer sitzt vor Terminal `"A"`, lässt sein Grafik ausgebendes Programm am Rechner `"B"` und das Programm `XXgrab` am Rechner `"C"` laufen, hat aber sowohl auf `"B"` als auch auf `"C"` die `DISPLAY`-Variable auf `"A:0.0"` gesetzt.

`call GrabName`
Das Programm fragt im Standard-Ausgabefenster nach einem Dateinamen für die zu schreibende(n) Datei(en). Es wird von `GrabInit` selbst aufgerufen, kann aber auch später zum Ändern des Filenamens erneut gerufen werden. Der Name sollte kurz sein, da `XXgrab` eine *laufende Nummer* und die Dateierweiterung `".gif"` automatisch anhängt.

`call GrabImage`
`XXgrab` wird „aufgefordert“, das durch `GrabInit` spezifizierte Fenster als Bitmap in seinen Arbeitsspeicher zu kopieren und anschließend als File mit nächster laufender Nummer zu schreiben. Das Fenster wird dazu auf dem Bildschirm nach oben geholt (mittels `GRaiseWin`). Bei Überlappung durch ein anderes Fenster wird der Inhalt nicht korrekt kopiert (also: *man gönne der Maus etwas Ruhe*).
Das Programm wartet, bis der Fensterinhalt kopiert wurde (meist weniger als eine Sekunde), danach darf die nächste Ausgabe ins Fenster erfolgen, während `XXgrab` noch mit dem Schreiben des Files beschäftigt ist (mehrere Sekunden, in Abhängigkeit von der Größe des Fensters).

`call GrabImageNr ( nr )`
Führt dieselbe Funktion wie `GrabImage` aus, mit dem Unterschied, dass die ganze Zahl `nr` zum Vervollständigen des Dateinamens verwendet wird anstelle einer laufenden Nummer (z. B. die Nummer des Zeitschritts bei einer zeitabhängigen Berechnung).

`call GrabDone` oder `call KillXXgrab`
Es wird ein Endesignal an `XXgrab` gesendet und dieses Programm damit beendet.

### XXplot - Gnuplot-Schnittstelle

Dieses Programm bietet die Möglichkeit, Daten unmittelbar vom laufenden (Parallelrechner-) Programm an GNU-Plot zur Darstellung zu übergeben. Die Verbindung zu `XXplot` wird durch folgende Unterprogramme realisiert:

`call Plot_Init`
Initialisiert die Verbindung zu `XXplot` in ähnlicher Weise wie dies durch `GrabInit` mit `XXgrab` geschieht (s.o.). Es wird lokal ein temporäres File angelegt, um die zu plottenden Werte abzuspeichern. Das Programm ist in der übersetzten Version für die Darstellung von bis zu 5 Funktionen $y_i = f_i(x)$ ausgelegt.
Durch wiederholten Aufruf von `Plot_Init` können die bisherigen Daten im temporären File überschrieben werden.

`call Plot_Name ( i, string )`
Definiert den Namen der $i$-ten darzustellenden Funktion für das Plotfenster von GNU-Plot. Ansonsten werden als Standard die Namen `"Dat1", ..., "Dat5"` verwendet.

`call Plot_Val ( x, n, y )`
Schreibt eine Zeile mit den Werten $x, y_i \;\; (i = 1, \ldots, n)$ in das temporäre File.
Das Programm vertraut darauf, dass die Zahl `n` bei jedem Aufruf gleich ist. Anderenfalls sind Fehlfunktionen möglich.

`call Plot_Show ( Titel, ShowAll )`
Sendet den aktuellen Inhalt des temporären Datenfiles an `XXplot` zur Darstellung der Funktionen in einem GNU-Plot-Fenster. Die Zeichenkette `Titel` wird als Überschrift verwendet.
`ShowAll` ist ein logischer Ausdruck mit folgender Bedeutung:
`.TRUE.`: alle Funktionen werden zugleich, in einunddemselben Koordinatensystem dargestellt (was bei sehr unterschiedlichen Wertebereichen nicht unbedingt eine gute Darstellung sein mag); das Programm läuft sofort weiter; (*Batch-Modus*)
`.FALSE.`: es erfolgt eine Abfrage, welche Funktionen dargestellt werden sollen, diese Abfrage wiederholt sich bis zur Eingabe einer 0 als Funktionsnummer (*interaktiver Modus*).

`call Plot_Cmd ( text )`
Sendet eine beliebige Zeichenkette an `XXplot`, die dort „ungefiltert“ an GNU-Plot als Kommando weitergegeben wird.
Hinweis: `XXplot` hat auch ein Bedienfenster, das neben den hier genannten Darstellungen noch Kommandoeingaben von Hand gestattet, (z. B. Achsenskalierung und ähnliches).

`call Plot_Close`
Dem Programm `XXplot` wird mitgeteilt, dass keine weiteren Anforderungen mehr folgen, woraufhin sich dieses vom Bildschirm entfernt.
Hinweis: Die temporäre Datei mit den Plot-Daten wird nicht automatisch gelöscht, steht also noch zur weiteren Auswertung zur Verfügung. Zu diesem Zweck gibt es ein kleines Programm `Plot`.

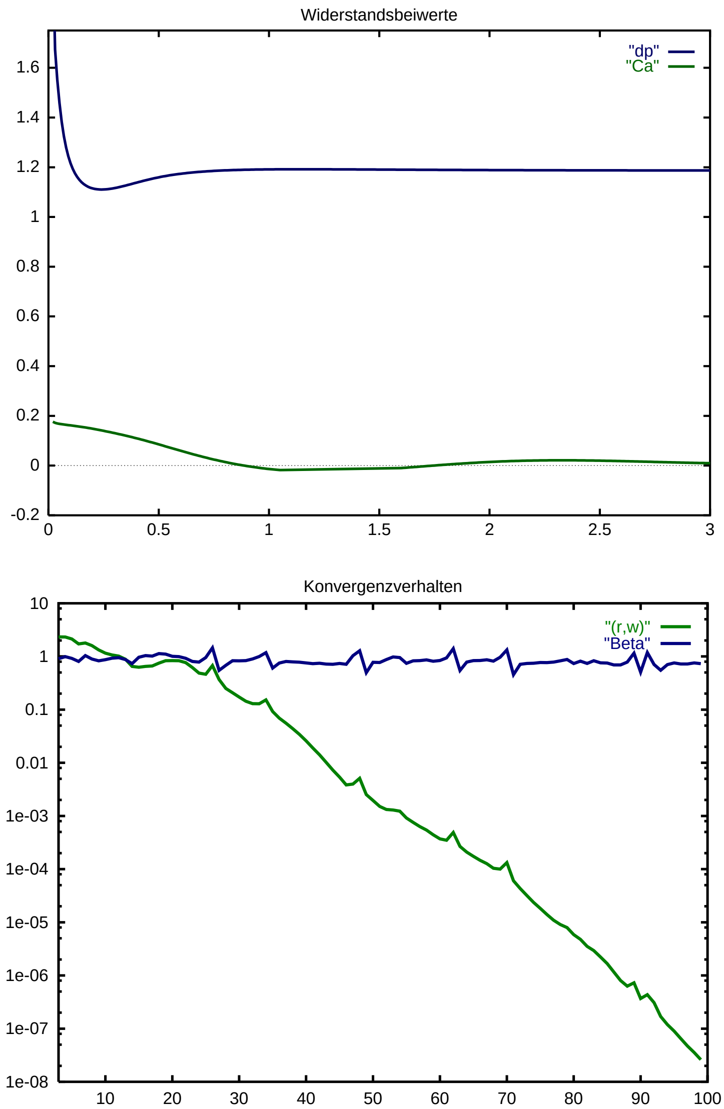


Abbildung 6: Beispiele zur Ausgabe mit `XXplot`

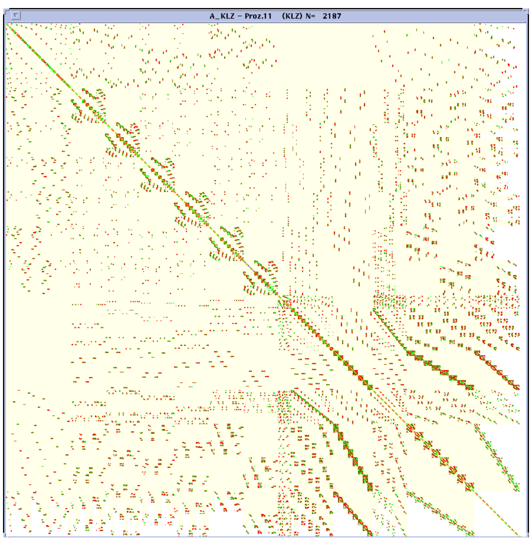

Abbildung 7: Besetzungsmuster einer Matrix (`GRKLZ`)

## GRKLZ- Besetzungsmuster von Sparse-Matrizen anzeigen

**Rufzeile:** call GRKLZ ( N, A, LA, TYP, HEADER )

**Parameter:**

- N - Dimension der Matrix
- A - Feld der Matrixelemente (DOUBLE PRECISION)
- LA - Leitvektor zu A
- TYP - Angabe der Speicherungsform von A als String; zulässig sind: 'KLZ','KZU','VBZ'.
- HEADER - Zeichenkette, die als Fenstertitel eingesetzt wird.

**Funktion:** Das Programm öffnet ein Fenster auf dem durch die Environment-Variable $DISPLAY angegebenen Bildschirm und stellt in diesem die Nichtnullelemente der Matrix A als Pixel dar. Die Darstellungsart kann durch Tastendruck im Fenster modifiziert werden:

- <SPACE> schaltet um zwischen komprimierter Darstellung (ganze Matrix im Fenster, mehrere Elemente ergeben ein Pixel) und expandierter Darstellung (pro Element genau ein Pixel). In letzterem Fall wird bei großen Matrizen nur ein Ausschnitt angezeigt.
- <+>,<-> Bei Ausschnittsanzeige kann man mit "+" vorwärts und mit "-" rückwärts blättern; es wird allerdings nur der Bereich um die Hauptdiagonale angezeigt, es ist keine Verschiebung nach rechts und links vorgesehen.
- <s> bei symmetrischen Matrizen (KLZ, VBZ) ist nur das obere Dreieck gespeichert. Mit "s" wird auf eine symmetrische Darstellung der vollen Matrix umgeschaltet und wieder zurück.
- <l> bei Kompaktlistenspeicherung (KLZ, KZU) werden nur die Nichtnullelemente angezeigt. Mit "l" wird der Hintergrund des Profils (entsprechend einer VBZ-Speicherung) hell unterlegt.
- <b> Umschalten zwischen Farb- und Schwarz-Weiß-Darstellung. Bei Farbdarstellung (nur KLZ/KZU) bedeuten die Farben:

  | | |
  |---|---|
  | rot | negative Matrixelemente |
  | grün | positive Matrixelemente |
  | gelb | Hauptdiagonalelemente |
  | blau | Oxer (spezielle Anzeiger auf der Diagonale, $> 10^{30}$ oder $< 10^{-18}$) |

  Bei komprimierter Anzeige (s.o.) ist diese Farbzuordnung nicht mehr eindeutig.
- <c> Sollten bei der Farbdarstellung Probleme mit der Sichtbarkeit der Farben auftreten, kann man hiermit dem Fenster eine private Farbpalette zuordnen.
- <q><br><ESC><br><ENTER> Beenden des Unterprogramms; Fenster wird geschlossen.

**Beispiel:** (s. Abb. 7)

# 5 Allgemeine Hilfsprogramme – Tools

Die hier aufgeführten Hilfsroutinen sind aus verschiedenen Anwendungen zusammengetragene Unterprogramme, die durchaus von allgemeinerem Interesse sein können, weil sie in dieser oder jener Form immer wieder auftauchen. Meist handelt es sich um Programme zur Ausgabe bestimmter Informationen in bestimmtem Format zu Testzwecken. Solche Ausgaben werden oft nicht nur deshalb als Unterprogramm realisiert, weil sie an verschiedenen Stellen des Programms auftreten, sondern weil gelegentlich in Fortran Felder nur als Platzhalter definiert sind und der vereinbarte Typ nicht mit dem gespeicherten Inhalt übereinstimmen muss. Zunächst aber einige Unterprogramme, die im Zusammenhang mit der Dialogprogrammierung nützlich sind.

`Integer Function LEN_TRIM ( string )`

Funktionswert ist die Länge der Zeichenkette ohne abschließende Leerzeichen, die in Fortran jeweils bis zur vereinbarten Länge einer Variablen angehängt werden. Diese Funktion gehört nicht bei allen Fortran-Installationen zum Standard, deshalb ist sie in dieser Bibliothek enthalten.

`Logical Function YES ( string )`

`Logical Function YES_P ( string )`

Die Zeichenkette erscheint auf der Standardausgabe, gefolgt von `" (J/N) - "` und es wird die Eingabe eines Zeichens erwartet. Die Funktion liefert den Wert `.TRUE.`, wenn eines der Zeichen `"J","j","Y","y"` eingegeben wurde, sonst `.FALSE.`.
Die Funktion `YES` ist als Einzelprozessorvariante vorgesehen, bzw. für den alleinigen Aufruf durch Prozessor 0 eines parallelen Programms. Die Funktion `YES_P` muss dagegen auf allen Prozessoren eines parallelen Programms gerufen werden. Sie benutzt intern die Kommunikationsroutinen zum globalen Austausch der Nutzereingabe. (Der Funktionsname `YES` bzw. `YES_P` muss im rufenden Programm explizit vereinbart werden!)

`Subroutine UPCASE ( string )`

Die in der Zeichenkette enthaltenen Kleinbuchstaben werden in Großbuchstaben umgewandelt.

`Subroutine BEEP ( n )`

Es werden `n` kurze Töne nacheinander ausgegeben (die auch zu einem etwas längeren Ton zusammengezogen sein können).

`Subroutine ENTER`

Prozessor 0 wartet auf eine beliebige (leere) Eingabe und veranlasst durch `Tree_Down` die Fortsetzung des Programms. – Auf allen Prozessoren aufzurufen !

`Subroutine FRCHAN ( log_unit )`

Es wird eine in diesem Moment noch nicht verwendete Dateinummer in `log_unit` zurückgegeben. Bei der Arbeit mit mehreren Dateien in einem Programm kann dieses Unterprogramm zur unentbehrlichen Hilfe werden, um versehentliche Doppelverwendung von logischen Dateinummern auszuschließen. Es sollte unmittelbar vor einer (jeden) `OPEN`-Anweisung gerufen werden, um die ermittelte Dateinummer dann für die zu öffnende Datei zu verwenden.

`Subroutine FLUSH ( log_unit )`

Fortran-Standardfunktion zum Entleeren des Puffers; wichtig nach einer Eingabeaufforderung und vor `READ`, falls das Programm über *remote shell* abgearbeitet wird:

```
call FLUSH(6)
```

`Subroutine OUTINT ( str, n, ifeld )`
: Ausgabe eines `Integer*4`-Vektors (`n` Komponenten) mit der Zeichenkette `str` als Überschrift.

`Subroutine OUTDBL ( str, n, dfeld )`
: Ausgabe eines `DoublePrecision`-Vektors (`n` Komponenten) mit der Zeichenkette `str` als Überschrift.

Es existieren weitere ähnliche Ausgaberoutinen für spezielle Datenstrukturen (Randketteninformationen, Knotenliste, Elementliste, hierarchische Liste, KLZ-Matrizen u. a.).

# 6 Nutzung der Bibliotheken

## 6.1 Wie werden die Bibliotheken eingebunden ?

Die Bibliotheken stehen für die unterschiedlichen an der Fakultät für Mathematik bisher verwendeten Entwicklungsumgebungen zur Verfügung.

| | | |
|---|---|---|
| Namen: | `libvbasmod.a` | Vektoroperationen (Abschn. 2), es handelt sich um eine der Varianten: `libvbasmodf.a` (Fortran-Quellen), `libvbasmodc.a` (C-Quellen), `libvbasblas.a` (C-Quellen inklusive maschinennah implementierter BLAS-Routinen) |
| | `libCubecom.a` | Kommunikationsbibliothek, unter diesem Namen meist auf der Basis von PVM (Abschn. 3) |
| | `libMPIcubecom.a` | MPI-Version von `libCubecom.a`, |
| | `libMPIcom.a` | alternativ zu `libCubecom.a`, benutzt spezifische MPI-Routinen, Prozessoranzahl muss keine Zweierpotenz sein |
| | `libNoPVM.a` | Dummy-Routinen zum Linken einer Workstation-Version (ohne PVM- oder MPI-Bibliothek) |
| | `libGraf.a` | Grafikroutinen (Abschn. 4) |
| | `libNoGraf.a` | Dummy-Version für einige Routinen der Bibliothek `libGraf.a`, damit ist ggf. (ohne Quelltextänderungen) das Linken ohne X11-Bibliothek möglich |
| | `libTools.a` | Hilfsroutinen, die z. T. in `libGraf.a` benutzt werden |
| | `libX11.a` | Standardbibliothek (zusätzlich zu `libGraf.a` notwendig) |
| Host : | `*.tu-chemnitz.de` | |
| Pfad : | `/afs/tucz/project/sfb393/FEM/libs/$archi/` | |

Sie ist in dem jeweiligen maschinen- bzw. systemabhängigen Unterverzeichnis zu finden, dessen Name zweckmäßigerweise auf einer Umgebungsvariablen (`archi`) gespeichert wird. Die hierfür benutzten Verzeichnisnamen sind:

| | |
|---|---|
| `archi` = `parix` | Parix für Transputersysteme |
| `archi` = `ppc` | Parix für PowerXplorer/GCPowerPlus |
| `archi` = `SUN4` | Sun-Workstation, SunOS |
| `archi` = `HPPA` | HP-Workstation, HP-UX |
| `archi` = `SGI64` | SGI-Workstation, IRIX 6.3 |
| `archi` = `SGIn32` | wie `SGI64`, Compiler-Option `-n32` (Application Binary Interface) |
| `archi` = `LINUX` | Linux für PC-Cluster, PVM oder MPICH |
| `archi` = `LINUX_lam` | Linux, LAM-MPI |
| `archi` = `CLIC` | wie `LINUX_LAM`, aber lokal auf den Knoten des CLiC |

Die Bibliotheken können somit durch Nutzung der folgenden Schalter eingebunden werden:

```
GRAF      = Graf
CUBECOM   = MPIcubecom
LINKFLAGS = \
    -L/afs/tucz/project/sfb393/FEM/libs/$(archi) \
    -l$(GRAF) -lTools [-lNoPVM] -l$(CUBECOM) -lvbasmod
```

wobei mit `-L` der Suchpfad[10] zur Einbindung der Bibliotheken spezifiziert wird und mit `-l` jeweils der Stamm des Bibliotheksnamens. Die Make-Variablen `GRAF` und `CUBECOM` zeigen, wie man bequem zwischen verschiedenen Bibliotheksvarianten wählen kann.

Einige *technische* Hinweise sollten beachtet werden:

- Bei manchen Compilern ist die Reihenfolge der Angabe der einzelnen Bibliotheken von Bedeutung, um alle Unterprogramme zu finden. In diesem Fall ist die obige Reihenfolge zu empfehlen, wenn alle genannten Bibliotheken verwendet werden. Wenn die Bibliothek `libNoPVM.a` verwendet wird, so muss diese <u>vor</u> der Kommunikationsbibliothek eingebunden werden.
- Auch für C-Programmierer, die obige Bibliotheksroutinen (in Fortran geschrieben) nutzen, ist es sinnvoll, das abschließende „Linken" des Programms mit Hilfe des Fortran-Compiler-Kommandos (z. B. `$(F77)` in Makefiles) durchzuführen, da der C-Compiler i. a. nicht die notwendigen Fortran-Standardbibliotheken benutzt und somit unaufgelöste Referenzen entstehen.
- Unter HP-UX sollte der Fortran-Compiler mit `fort77` statt mit `f77` aufgerufen werden, sonst können die Compilerflags `-L` und `-l` nicht in gewohnter Weise verwendet werden.

## 6.2 Aufruf der Bibliotheksroutinen in C-Quellen

Es sind in jedem Fall die *üblichen* Regeln für die Verbindung von Fortran- und C-Programmen zu berücksichtigen. Da die Programme mit Blick auf eine Nutzung innerhalb von F77-Quellen geschrieben wurden, sind zwei Dinge beim Aufruf aus C-Quellen heraus zu beachten.

1. **Alle** Parameter der Rufzeile müssen als Pointer übergeben werden (call by reference).
   **Bem.:** Bei Zeichenkettenparametern wird durch den Fortran-Compiler jeweils ein zusätzlicher Parameter für die Länge der Zeichenkette an die *offizielle* Parameterliste angehängt, dieser wird nicht als Pointer sondern als Wert übergeben. Die Zeichenkette selbst enthält <u>**kein**</u> spezielles Endekennzeichen (Nullbyte), wie es etwa in C üblich ist.[11] So wird der auf Seite 33 definierte Unterprogramm-Aufruf `call GStoreName(iwin,title)` in C z.B. so realisiert:

   ```
   int iwin;
   char* title = "My solution";
   ...
   gstorename ( &iwin, title, strlen(title) );
   ```

2. Der Rufname muss den Konventionen der jeweiligen Hard- und Software-Umgebung genügen. Es gelten (leider nicht einheitliche) Regeln zur Namensbildung in C für

[10] Die Schreibweise `$(archi)` innerhalb des Makefiles entspricht der Umgebungsvariablen `$archi`

[11] Nullterminierte Strings können zwar in Fortran mittels Compilerswitch ebenfalls verwendet werden, aber aus Kompatibilitätsgründen innerhalb verschiedener Fortran-Versionen wird oft darauf verzichtet.

Fortran-Programmnamen (und Namen von Common-Blöcken).
Die Konventionen für unsere bisher betrachteten Anwendungsfälle sind:

| | | |
|---|---|---|
| Parix (Transputer) | – | Kleinbuchstaben mit angehängtem Unterstrich "_" |
| Parix (PowerPC) | – | Kleinbuchstaben |
| SunOS | – | Kleinbuchstaben mit angehängtem Unterstrich "_" |
| HP-UX, SGI | – | Kleinbuchstaben |
| nCUBE | – | Großbuchstaben |
| LINUX (f2c bzw. g77) | – | Kleinbuchstaben mit angehängtem Unterstrich "_", wenn der Fortran-Name selbst schon ein "_" enthält, so werden am Ende zwei angehängt: "__". |
| Linux (NAG bzw. Intel) | – | Kleinbuchstaben mit angehängtem Unterstrich "_" |

Um die Quelltexte weitgehend von solchen Besonderheiten der Fortran-C-Schnittstelle zu befreien, werden die Namen innerhalb von Header-Files als Makro definiert, z. B.:

```
#ifdef UNDER          /* Unterstrich am Ende */
#define   dscapr        dscapr_
#define   vdaxpy        vdaxpy_
#ifdef DBL_UNDER      /* hier sogar zwei     */
#define   my_prog       my_prog__
#else
#define   my_prog       my_prog_
#endif    /* LINUX */
#endif    /* UNDER */

#ifdef CAPS           /* alles in Grossbuchstaben */
#define   dscapr        DSCAPR
#define   vdaxpy        VDAXPY
#define   my_prog       MY_PROG
#endif    /* CAPS  */
```

Mit entsprechenden Compileroptionen `-DUNDER` (und `-DDBL_UNDER`) oder `-DCAPS` wird die jeweils richtige Version ausgewählt, anderenfalls steht der jeweilige Name für sich selbst.

Solche Headerfiles mit Definition einheitlicher Namen (jeweils in Kleinbuchstaben) und ggf. Prototypen können mittels `-I/afs/tucz/project/sfb393/FEM/libs/headers` verwendet werden:

| | | |
|---|---|---|
| `<vbasmod.h>` | - | für die Vektorroutinen aus `libvbasmod.a` |
| `<trnet.h>` | - | für die Kommunikationsroutinen aus `libCubecom.a` |
| `<ggnames.h>` | - | für die X11-Grafikroutinen aus `libGraf.a` |

Zur Illustration diene die C-Version der Routine `multblqs` (Abschnitt 2.4.3, S. 7), 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` ersetzt werden, welches auch die entsprechenden Makrodefinitionen aller Unterprogrammnamen der Bibliothek enthält.

```
/*****************************************************************/
/**     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                                 */
/*****************************************************************/

/*   Abbildung des Namens durch Makrodefinition, die mittels     */
/*   Compiler-Optionen gesteuert werden kann, z.B. -D UNDER      */
#ifdef UNDER
#define multblqs    multblqs_
#endif
#ifdef CAPS
#define multblqs    MULTBLQS
#endif

#include <vbasmod.h>       /*  mit Makrodefinitionen dscapr, vdaxpy  */
                           /*  und Typvereinbarungen                 */
                           /*     double dscapr ();                  */
                           /*     void   vdaxpy ();                  */

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;

/*                                        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);
}
```

Das Headerfile `trnet.h` enthält auch die Definitionen für den Zugriff auf die unter 3.1 genannten globalen Variablen aus den Fortran-Common-Blöcken. Diese Variablen können damit im C-Quelltext wie in Fortran verwendet werden (linke Spalte der Tabelle, Namen komplett in Großbuchstaben) oder auch in der Originalschreibweise als Komponenten von Strukturvariablen (rechte Spalte der Tabelle):

| Fortran oder C | Alternative in C |
|---|---|
| `NCUBE` | `trnet.ncube` |
| `ICH` | `trnet.ich` |
| `NPROC` | `trring.nproc` |
| `ICHRING` | `trring.ichring` |
| `LFORW` | `trring.lforw` |
| `LBACK` | `trring.lback` |

Auch hierzu zeigt ein Beispiel die typische Vorgehensweise:

```
#include <stdio.h>
#include <trnet.h>
#include <vbasmod.h>
#define   LANG     10000

void main()
{
   int           istart,  nlocal = LANG,
                 i, nsum, lang, mode = 0;
   double        x[LANG],  y[LANG],
                 help,     sum;

   trinit (&mode);         /* Initialisiere Hypercube      */
   nlocal = LANG;
   nsum   = 1;
   lang   = LANG;
   istart = 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 ( ICH == 0 )
      printf("result : %lf\n", sum);

   trclose ();                 /* Paralleles Programm beenden */
}
```

## 6.3 Nutzung unter PVM

Für mehr oder weniger heterogen zusammengesetzte Workstation-Cluster ist PVM ( = **P**arallel **V**irtual **M**achines, [10] ) weit verbreitet (bereits vor der Entwicklung von MPI). Als einzige der hier dokumentierten Bilbiotheken ist `libCubecom.a` speziell an die verwendete Kommunikationsbasis (PVM, MPI, Parix) angepasst, indem zur Realisierung der elementaren Kommuniaktionen die entsprechenden Unterprogramme benötigt werden. Beim Linken für PVM sind deshalb zusätzlich die Bibliotheken `libfpvm3.a` und `libpvm3.a` erforderlich.

In der PVM-Version 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 zu einem großen Teil in der Verantwortung des PVM-Daemons (`pvmd`) liegt. Es sind auch mehrere Prozesse auf derselben Maschine möglich, solange der Speicher reicht.

Die Routine `trinit` wurde so implementiert, dass die Prozesse nach Möglichkeit gleichmäßig auf die verfügbaren Hosts innerhalb des Workstationclusters verteilt werden (PVM selbst ist dazu oft nicht in der Lage).

Die unter PVM vorgesehene heterogene Struktur der Maschinenarchitekturen hinsichtlich unterschiedlicher binärer Datenformate sowohl bei `Integer`- als auch bei `Real`-Daten wird hier nicht unterstützt, d.h. Daten werden ohne jede Konvertierung binär gesendet. Von `trinit` wird die Homogenität des Clusters zu Beginn überprüft. Allerdings sind auch unterschiedliche Workstations, z. B. vom Typ Sun und HP, in diesem Sinne miteinander verträglich. Dagegen lassen sich diese Workstations und Linux-PC nicht in einem Cluster gemeinsam betreiben.

Die Nutzung von PVM setzt eine bestimmte einheitliche Verzeichnisstruktur und natürlich ein Login auf allen beteiligten Maschinen voraus, möglichst mit der Fähigkeit, *remote-shell*-Kommandos auszuführen (`.rhosts`-File) bzw. *ssh*-Kommandos. Im Homeverzeichnis sollte das Unterverzeichnis `pvm3` existieren, auf das bei jedem Login (z. B. in `.cshrc`) eine Environment-Variable gesetzt wird:

```
setenv PVM_ROOT ~/pvm3
```

Das ist erforderlich, um den PVM-Daemon erfolgreich starten zu können. Zum Starten von PVM sei auf das Manual verwiesen (z. B. `man pvm` und `man pvmd3`).

Das Verzeichnis `pvm3` selbst sollte folgenden Inhalt besitzen:

```
~/pvm3/lib/        --> symb. Link auf: /afs/tucz/project/sfb393/pvm3/lib
       include/    --> symb. Link auf: /afs/tucz/project/sfb393/pvm3/include
       bin/SUN4/       \
           HPPA/        >  Verzeichnisse fuer die eigenen Programme
           LINUX/      /
           ...
```

Die Namen der architekturabhängigen Unterverzeichnisse sind von PVM vorgegeben und können mit dem Kommando

```
$PVM_ROOT/lib/pvmgetarch
```

für den jeweiligen Rechner angezeigt werden.

Das Programm, das gerade abgearbeitet werden soll, muss auf jedem Rechner des PVM-Clusters unter demselben Namen im jeweiligen Verzeichnis

```
$PVM_ROOT/bin/‘pvmgetarch‘/
```

(also z. B. `~/pvm3/bin/SUN4/myhelloworld` und `~/pvm3/bin/HPPA/myhelloworld`) zu finden sein. Die Realisierung der `Cubecom`-Routinen für PVM (z.Z. Version 3.4) ermöglicht auch den Test der Programme auf nur einer Maschine (`NPROC=1`), ohne dass der PVM-Daemon läuft.

Eine Einzelprozessor-Version mag im Zusammenhang mit parallelen Algorithmen unsinnig erscheinen. Aber wegen der Besonderheit der Kommunikationsroutinen, auch im „entarteten“ Hypercube (Dimension 0) korrekt zu funktionieren, kann ein für Parallelrechner geschriebenes (bezüglich der Prozessoranzahl skalierbares) Programm bei Bedarf auch auf einem beliebigen anderen Rechner als Ein-Prozessor-Variante getestet werden. Es muss lediglich ein Programm `TRINIT` existieren, das die wesentlichen Variablen belegt: `NCUBE = 0, NPROC = 1, ICH = 0`. Die `SEND`- und `RECV`-Unterprogramme brauchen nur Dummy-Routinen zu sein, da sie zwar als Rufzeile in den Programmen enthalten sind, aber nie aufgerufen werden.

Solche Dummy-Routinen sind in der Bibliothek `libNoPVM.a` enthalten. Bei Verwendung dieser Bibliothek (vor `libCubecom.a`) werden die speziellen PVM-Bibliotheken zum Linken nicht benötigt.

## 6.4 Nutzung unter MPI

In den letzten Jahren hat sich MPI als Standard für ein Message Passing Interface etabliert. Nichtsdestotrotz gibt es unterschiedliche und miteinander nicht verträgliche Implementierungen dieses Standards. Die besonders verbreiteten (nicht auf spezielle Architekturen zugeschnittenen) MPI-Implementierungen sind MPICH und LAM-MPI, jeweils in mehreren und immer wieder neuen Versionen. Die Unverträglichkeit der verschiedenen MPI-Versionen bezieht sich darauf, dass es nicht genügt, eine andere MPI-Bibliotheken beim Linken zu benutzen. Man muss die eigenen Programme neu übersetzen, sofern sie Unterprogramme des „MPI-Standards“ aufrufen. Ebenso bringt jede MPI-Version ihre eigene Laufzeitunmgebung mit (Daemons, mpirun).

Aus diesen Gründen gibt es mittlerweile neben unserer Bibliothek `libCubecom.a` alternative Bibliotheken. Somit kann der Nutzer die zu seinem gewünschten Kommunikationssystem passende `Cubecom`-Bibliothek benutzen. Ohne Quelltextänderungen lässt sich damit ein Programm mit verschiedenen Systemen wie PVM, MPICH, LAM-MPI testen.

Diese Alternativen der `Cubecom`-Bibliothek sind unter folgenden Bezeichungen in den bereits genannten Bibliotheksverzeichnissen enthalten (für die Rechnerarchitekturen ohne Parix, speziell `archi=LINUX`, bzw. `archi=CLIC`):

`libCubecom.a` – Kommunikationsroutinen benutzen PVM (Version 3.4)

`libMPIcubecom.a` – Kommunikationsroutinen benutzen MPI, i. allg. sollte das die Implementierung LAM-MPI sein. Es werden im wesentlichen nur die MPI-Routinen zur Kommunikation zwischen je zwei Prozessoren benutzt. Der globale Datenaustausch beruht weiterhin auf der (virtuellen) Hypercube-Struktur.

`libMPIcom.a` – Kommunikationsroutinen benutzen MPI. Anders als bei den vorgenannten Alternativen funktioniert diese Bibliothek mit beliebiger Anzahl von Prozessoren, indem hier die die speziellen MPI-Routinen für globale Kommunikation genutzt werden.

`libMPICHcom.a` – wie `libMPIcom`, aber mit Nutzung der MPICH-Implementierung.

`libMPICHcubecom.a` wie `libMPIcubecom`, aber mit MPICH.

## 6.5 Unterstützung für Makefiles

Unsere Bibliotheken sind für zahlreiche Rechnerarchitekturen bzw. Betriebssysteme nutzbar. In der Regel gibt es (leider) zwischen den Systemen erhebliche Unterschiede wie die Namen der Compiler und insbesondere die zu verwendenden Flags. Deshalb enthalten die systemspezifischen Bibliotheksverzeichnisse je eine Datei `default.mk`, die vom Nutzer ins eigene Makefile integriert werden kann, z.B. durch:

```
LIBHOME  = /afs/tucz/project/sfb393/FEM/libs
PARLIBS  = $(LIBHOME)/$(archi)
include $(PARLIBS)/default.mk
...
```

Die Make-Variable `$(archi)` wird aus der Umgebungsvariablen `$archi` übernommen. Sie kann jedoch auch unmittelbar als Argument von `make` angegeben werden, z.B.: `make archi=LINUX_lam`

Die Include-Datei `default.mk` definiert u. a. folgende Variablen:

| | |
|---|---|
| `F77` | Name des Fortran-Compilers (ggf. mit vollem Pfad), z.B. `g77` oder `$(MPIHOME)/bin/mpif77` |
| `C` und `CC` | Name des C-Compilers (beide Variablen sind i.allg. identisch belegt) |
| `FFLAGS` | typische Compilerflags für den Fortran-Compiler |
| `CFLAGS` | typische Compilerflags für den C-Compiler |
| `CPPFLAGS` | Flags für den C-Preprocessor, z.B. `-DUNDER` usw. |
| `MPILIB` | Linker-Optionen zum Einbinden der MPI-Bibliothek (falls notwendig; `MPILIB` ist z.B. leer, wenn `F77=mpif77`) |
| `PVMLIB` | Linker-Optionen zum Einbinden der PVM-Bibliothek |
| `AR` | Name des Archivators für Bibliotheken |
| `ARFLAGS` | Flags zum Archivieren von Objektfiles in einer Bibliothek |
| `RANLIB` | Kommando zum Aktualisieren der Symboltabelle in einem Bibliotheksarchiv, (meist nur ein `echo`-Kommando, da `$(AR)` diese Funktion selbst ausführt, Ausnahme z.B. Sun) |

Darüber hinaus werden einige Standardregeln wie `.f.o` definiert.

Wegen der teilweise unterschiedlichen Syntax innerhalb eines Makefiles hinsichtlich der Aktualisierung von Beständen einer Bibliothek stehen auch hierfür architekturspezifische Include-Dateien zur Verfügung. Der dabei berücksichtigte Unterschied besteht darin, dass bei manchen Systemen (Sun, LINUX) die Aktualisierungszeiten der Quellen mit denen der Bestände in der Bibliothek verglichen werden können, während bei anderen (HPPA, SGI, ppc) nur mit dem Aktualisierungsdatum der Bibliotheksdatei verglichen wird[12].

Die Include-Dateien werden mittels `include $(PARLIBS)/LIBMAKE...` ins eigene Makefile übernommen.

| | |
|---|---|
| `LIBMAKE` | Standarddefinitionen |
| `LIBMAKEf` | Zusatzdefinitionen für Fortran-Quellen (`*.f`) |
| `LIBMAKEF` | Zusatzdefinitionen für Fortran-Quellen, die mittels C-Preprocessor vorzubehandeln sind (`*.F`) |
| `LIBMAKEc` | Zusatzdefinitionen für C-Quellen |

Diese Include-Dateien können benutzt werden, wenn genau eine Bibliothek durch dieses Makefile erzeugt bzw. aktualisiert werden soll. Dazu müssen die folgenden Variablen im Makefile definiert sein:

| | |
|---|---|
| `LIBDIR` | Verzeichnis, in dem die Bibliothek steht (Variable kann leer sein, wenn sich die Bibliothek im aktuellen Verzeichnis befinden soll) |
| `stamm` | Basis des Bibliotheksnamens, dieser lautet dann `lib$(stamm).a` |
| `fsource` | Liste der Fortran-Quellen (`*.f`) |
| `Fsource` | Liste der Fortran-Quellen (`*.F`) |
| `csource` | Liste der C-Quellen (`*.c`) |
| `includes` | Liste der Include-Files, die von Fortran-Quellen benutzt werden (falls deren Änderung eine Neuübersetzung erzwingen soll) |

Es ist darauf zu achten, dass nur diejenigen Zusatzdefinitionen `LIBMAKEx` verwendet werden, für die auch die entsprechende Variable `xsource` (`x=f,F,c`) nichtleer definiert wurde.

[12] Letzteres kann dazu führen, dass nach einem unvollständigen `make`, z.B. wegen Syntaxfehler in einem der Programme, ein nachfolgendes `make` keine Aktualisierung älterer Bestände mehr vornimmt.

# Literatur

[1] G. Haase, T. Hommel, A. Meyer and M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC 93_3, TU Chemnitz-Zwickau, Mai 1993.

[2] 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.

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

[4] M. Meyer. Grafik-Ausgabe vom Parallelrechner für 3D-Gebiete. Preprint SPC 95_5, TU Chemnitz-Zwickau, Januar 1995.

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

[6] M. Pester. Grafik-Ausgabe vom Parallelrechner für 2D-Gebiete. Preprint SPC 94_24, TU Chemnitz-Zwickau, November 1994.

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

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

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

[10] V. S. Sunderam. PVM: A framework for parallel distributed computing. Technical report, Oak Ridge National Laboratory, 1992. `http://www.epm.ornl.gov/pvm/pvm_home.html`.