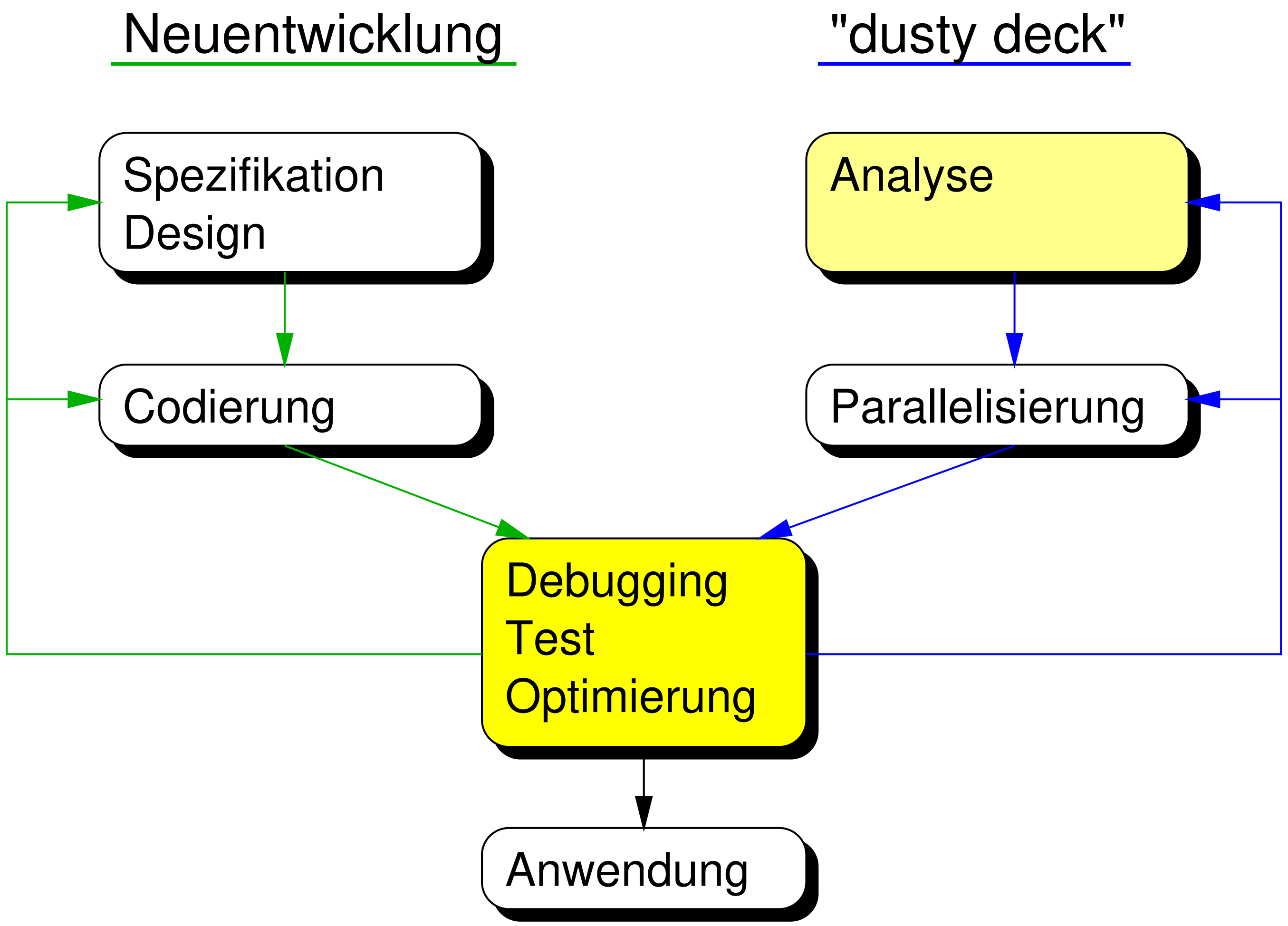
Werkzeuge zur parallelen Programmierung: Debugging und Visualisierung

Roland Wismüller

LRR-TUM
Institut für Informatik
Technische Universtität München
e-mail: wismuell@informatik.tu-muenchen.de

Inhaltsübersicht

Entwicklungszyklus paralleler Programme



Werkzeuge für Spezifikation und Design

Formale Methoden: z.B. FOCUS

- Systemkomponenten spezifiziert als stromverarbeitende Funktionen
- Eigenschaften beschrieben durch logische Prädikate über Strömen
- Gesamtsystem entsteht durch Kompositionsregeln

Graphische Methoden: z.B. Hamlet, Trapper, Hence

- Graphische Spezifikation der Prozeß- und Kommunikationsstruktur
  - $\rightsquigarrow$ Coderahmen für die Implementierung
  - $\rightsquigarrow$ Simulation des Systems (Hamlet)
- Darstellung von Laufzeitinformation in der Spezifikation
- Probleme: Replikation, Dynamik, Konsistenz der Implementierung

Werkzeuge für die Codierung

Parallele Programmierbibliotheken (PVM, MPI, BLAS, ...)

Codegeneratoren in Designwerkzeugen

Parallele Sprachen

- High Performance Fortran (HPF)
- Parallele Sprachen auf CSP-Basis
- Parallele objektorientierte Sprachen
  Objekte im System verteilt, Spracherweiterungen für Plazierung und Synchronisation, Zugriff transparent

Werkzeuge für Analyse und Parallelisierung

Interaktive Werkzeuge (z.B. Forge, MIMDizer)

- Variablen-Browser: Definition/Benutzung von (globalen) Variablen (Common-Blöcke / Equivalence)
- Programmstruktur: Prozeduraufrufgraphen ...
- Schleifenanalyse: Abhängigkeiten zwischen Schleifeniterationen

⇒ Datenaufteilung, Code-Umstrukturierung, Übersetzerdirektiven

Automatisch parallelisierende Compiler

- (V)LIW-Compiler (feingranularer Parallelismus)
- Compiler für Systeme mit gemeinsamem Speicher
  ⇝ Schleifenparallelität, Aufteilung auf Threads und Synchronisation
- Compiler für Systeme mit verteiltem Speicher
  ⇝ Datenaufteilung (Forschungsthema!)

Debugging: eine Definition

Debugging: Aufspüren und Beseitigen von Softwarefehlern

4 Schritte:

1. Test, Regressionstest
   (Re-)Produzieren eines Fehlers
2. Erkennen der Fehlerwirkung
3. Schließen von Fehlerwirkung auf Fehlerursache
4. Beseitigen des Fehlers

```
    . . .
    foo(a,b,x,&result)
    . . .
}
```

Ursache: vergessenes '&'

```
void foo(int a, int b, int *x, int *result)
{
    . . .
    *x = a+b;
    . . .
}
```

Wirkung: core dump
(segmentation violation)

Häufigste Fehlerquellen

Sequentielle Programmierung

1. Schnittstellenprobleme (Typen, Speichern von Zeigern auf Parameter)
2. Zeiger und dynamische Speicherverwaltung
3. Logische und algorithmische Fehler

Parallele Programmierung

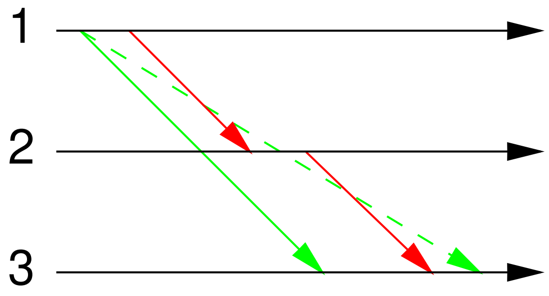
1. Kommunikationsfehler (Protokolle)
2. Races
3. Deadlocks

Races

Definition: Ein *Race* entsteht durch unsynchronisierte, modifizierende Zugriffe auf ein mehreren Prozessen gemeinsames Objekt.

Beispiel: „Dreiecks-Ungleichung“

Prozeß 1	Prozeß 2	Prozeß 3
send(3) send(2)	recv(1) send(3)	recv(-1)



Konsequenz: Nichtdeterminismus, nicht reproduzierbare Abläufe des parallelen Programms

Deadlocks

Definition: Bei einem *Deadlock* (Verklemmung) wartet eine Menge von Prozessen blockierend auf Ereignisse, die nur von den blockierten Prozessen selbst ausgelöst werden könnten.

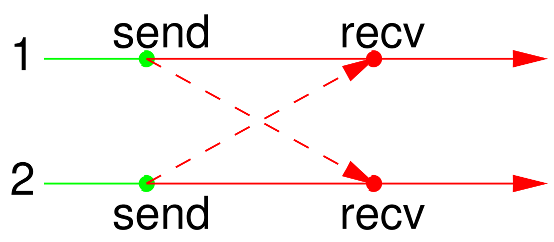
Beispiel: Datenparalleles, SIMD-artiges Programm

Prozeß 1

```
MPI_Send(...,2,...)
MPI_Recv(...,2,...)
```

Prozeß 2

```
MPI_Send(...,1,...)
MPI_Recv(...,1,...)
```



Konsequenz: Programm terminiert nicht.

Problem: Auftreten von Deadlocks u.U. abhängig von der Programmierbibliothek.

Werkzeugunterstützung für das Debugging

Übersicht:

1. Statische Analyse
2. `printf` und `WRITE`
3. Laufzeit-Debugger
4. Spur-basierte Werkzeuge
5. Haltepunkt-basierte Debugger

Statische Analyse

Analyse des Programmtextes vor/zur Übersetzungszeit

Sequentielle Aspekte:

- Strikte Typ- und Parameterprüfung
- Erweiterte semantische Tests (vollständige `switch`-Anweisungen, konstante Bedingungen, ...)

⤳ spezielle Werkzeuge: `lint`, `flint`, `insight`

⤳ gute ANSI-Compiler, Option `-Wall` (alle Warnungen)

Parallele Aspekte:

- Erkennung möglicher *Races*
- Prüfung auf Deadlock-Freiheit

⤳ Forschungsthema!

`printf` und `WRITE`

***Das* Debugging-Werkzeug schlechthin ...**

Aber bei parallelen Programmen:

- Zuordnung der Ausgaben zu den einzelnen Prozessen schwierig.
  Je nach Plattform zeilen- oder sogar zeichenweise Mischung möglich!

- Bei Programmierung auf Workstation-Clustern: Umleiten der Ausgabe in ein gemeinsames Fenster?!

$\leadsto$ *parallel print* (PTOOLS-Projekt)

$\leadsto$ `xpvm`: spezielle Sicht „Task Output“

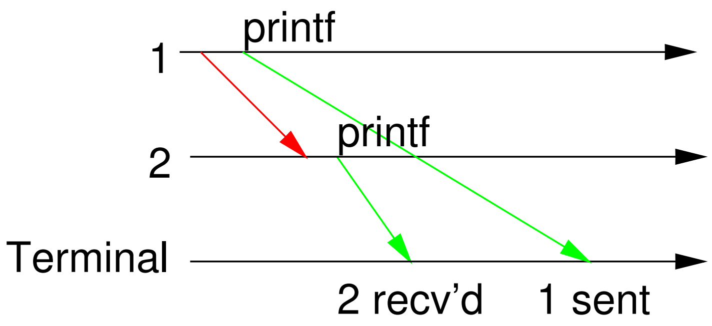
- Korrekte kausale Ordnung der Ausgaben ist nicht gewährleistet:

```
Prozeß 1

MPI_Send(...,2,...)
printf("1 sent\n")
```

```
Prozeß 2

MPI_Recv(...,1,...)
printf("2 recv'd\n")
```



Laufzeit-Debugger

Automatische Fehlerprüfungen zur Laufzeit

Sequentielle Aspekte:

- Dynamische Speicherverwaltung (z.B. `purify`, `insight`)

Parallele Aspekte:

- „Gute“ Programmierbibliothek (Parameterprüfungen ...)
- *Race*-Erkennung (Forschungsthema!)

Vorbereitung der Anwendung:

- Präprozessor und Neuübersetzung
- Binden mit speziell instrumentierten Bibliotheken
- Instrumentierung der Binärdatei

Spur-basierte Werkzeuge

Merkmale:

- Aufzeichnung relevanter Ereignisse während des Programmlaufs
- Betrachtung der Ereignisspur (*trace*) durch „Browser“
  - Off-line (nach Programmende) aus Spurdatei
  - On-line (während Programmlauf) über *pipe*
- Im Prinzip: Automatisierung der `printf`-Methode
- Nur in parallelen Umgebungen verbreitet (Portabilität)

Aufgezeichnete Ereignisse:

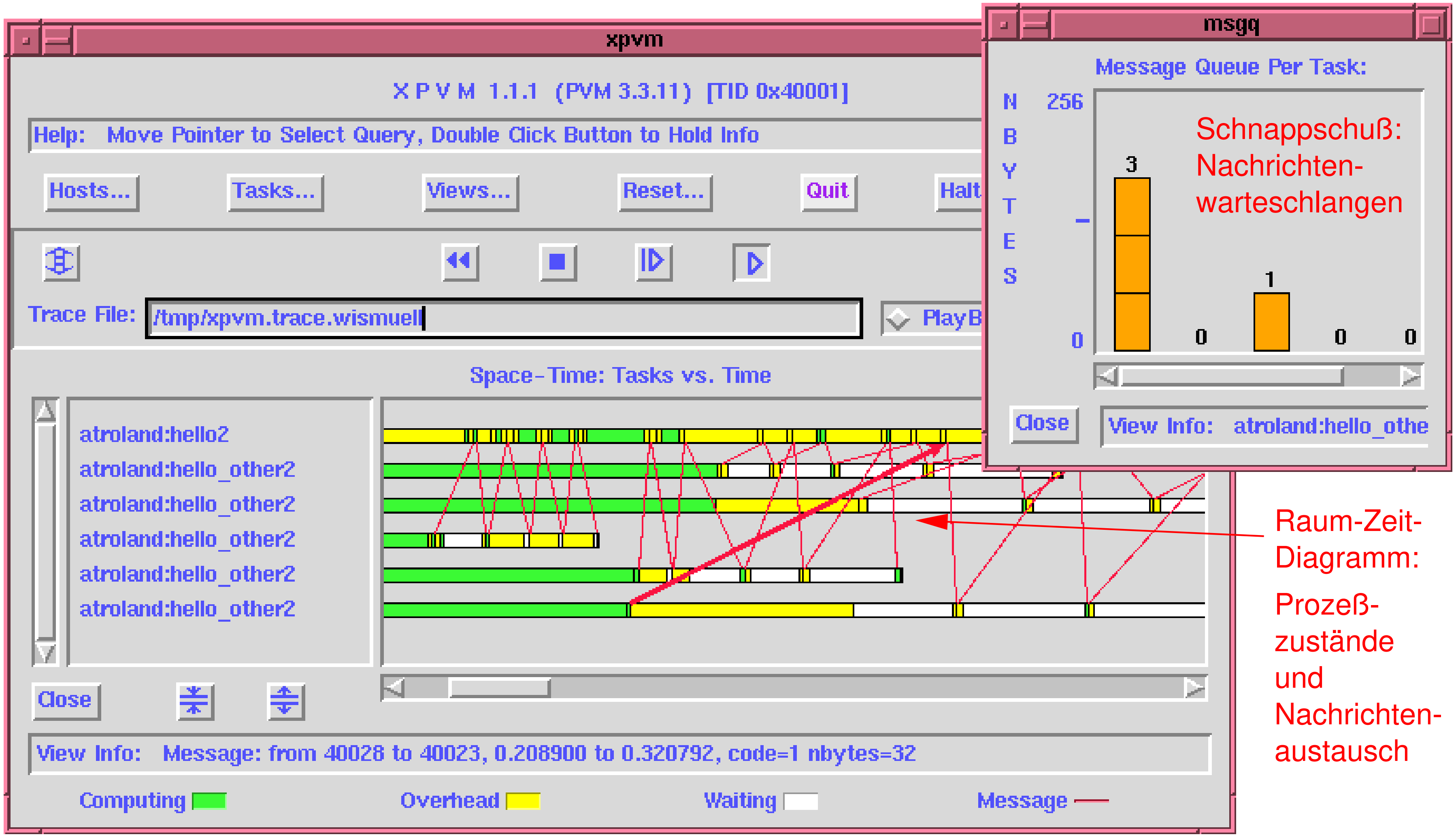
- Im allgemeinen Aufrufe von / Rückkehr aus Funktionen der Programmierbibliothek
- Nur zum Teil auch Quellcodebezug:
  - Aufrufposition von Bibliotheksfunktionen
  - Eintritt in / Austritt aus Benutzerfunktionen
  - Benutzerdefinierte Ereignisse

Spur-basierte Werkzeuge (2)

Darstellungsarten:

- Raum-Zeit-Diagramme, Gantt-Diagramme und Kombinationen
  - Darstellung der einzelnen Prozeßzustände bzw. -interaktionen über der Zeit
  - Z.B. `xpvm`, `xmpi`, `paragraph`, `trapper`
  - Zum Teil knoten- und prozeßorientierte Darstellung
  - \+ Globaler Überblick
  - − Spiegelt oft globale Zeit bzw. globale Ordnung vor, die nicht gewährleistet wird.
- (Animierte) Folge von Schnappschüssen
  - Darstellung des Globalzustands zu bestimmten Zeitpunkten
  - Z.B. `vistop`, `xpvm`, `xmpi`, `paragraph`
  - Meist verschiedene Sichten, z.B. Prozeß-/Knoten-Status, Kommunikation, Nachrichtenwarteschlangen, ...
  - \+ Detailliertere Darstellung des Zustands

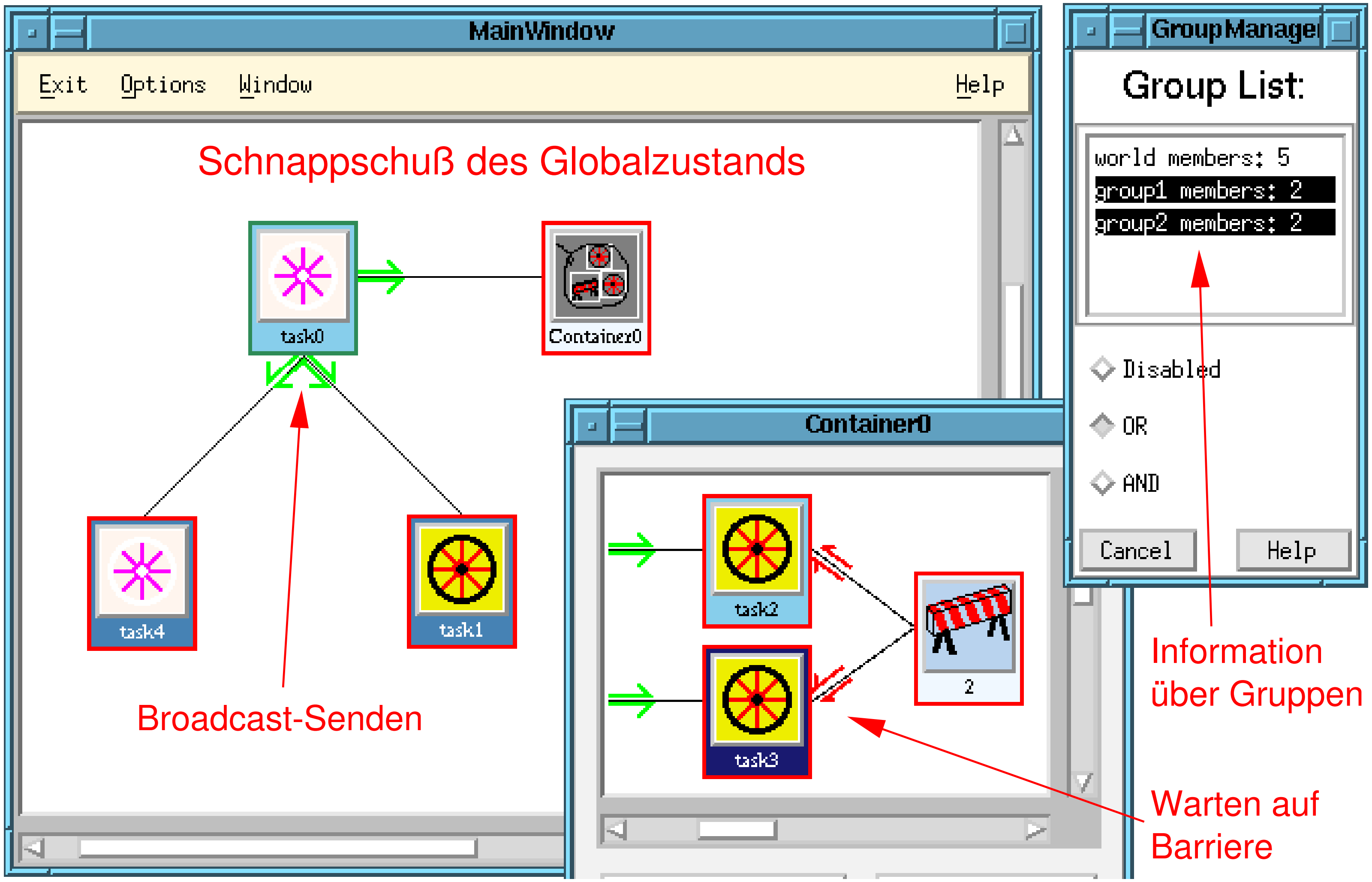
Spur-basierte Werkzeuge (3): Beispiel XPVM



Raum-Zeit-Diagramm:

Prozeß-zustände und Nachrichten-austausch

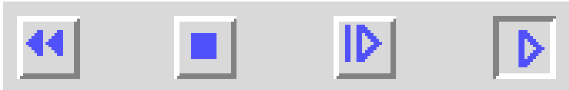
Spur-basierte Werkzeuge (4): Beispiel VISTOP



Spur-basierte Werkzeuge (5)

Steuerung:

- VCR-ähnliche Kontrollelemente:
  Start, Stop, Einzelschritt, Rückspulen,
  manchmal Rückwärtsschritt



- Zum Teil Auswahl und Positionierung der zu überwachenden Knoten oder Prozesse (vor oder während der Laufzeit)

Vorbereitung:

- Präprozessor und Neuübersetzung (v.a. bei Werkzeugen mit Quellcode-Bezug)
- Binden mit instrumentierten Bibliotheken
- Laufzeit-Option der Programmierbibliothek (z.B. bei PVM)

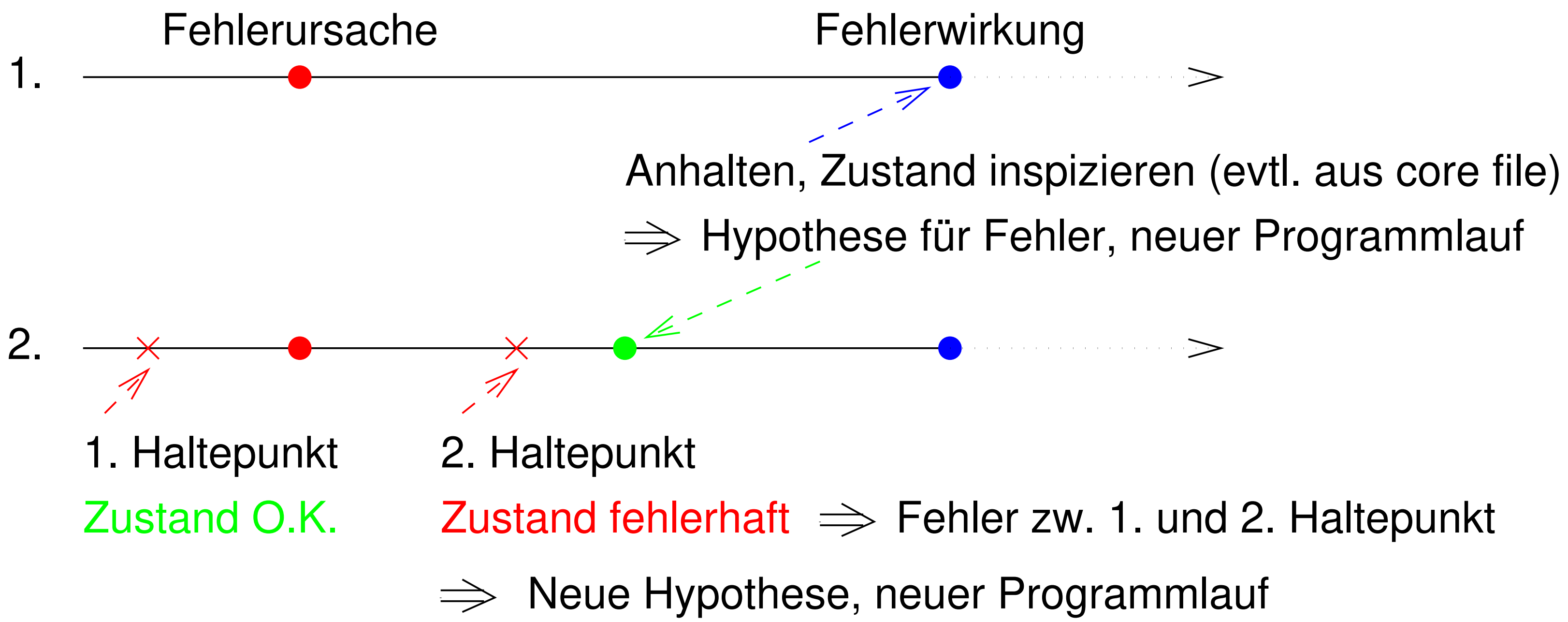
Bewertung:

- Für globalen Überblick über den Programmablauf
- Zur Überwachung von Kommunikation

Haltepunkt-basierte Debugger

Vorgehen:

- Anhalten des Programms an interessanten Stellen
- Inspizieren des Programmzustands
- Fortführung (evtl. Neustart) des Programms



Haltepunkt-basierte Debugger (2)

Typischer Funktionsumfang (sequentielle Debugger z.B. `gdb`, `dbx`):

- Anhalten des Programms
  - Bei Erreichen einer gegebener Zeile oder Funktion
    `break in main` oder `stop at foo.c:112`
  - Bei Änderung einer gegebener Variablen (langsam!)
    `watch (p == NULL)`
  - Kombinationen: bedingte Haltepunkte
    `break at 112 if (p == NULL)`
- Inspizieren des Programmzustands
  - Prozedur-Aufrufkeller
    ```
    where    #0  0xef773ec4 in exit ()
             #1  0x11eb0 in main (argc=1, argv=0xeffff5e4) at main.c:56
    ```
  - Parameter, Variablen
    `print p->str.next[i]`
    Verschattete Variablen durch `up` und `down` oder Spezifikation der aktiven Prozedur:
    `print ‘‘globalp->str.next[‘main.c:main‘i]`

Haltepunkt-basierte Debugger (3)

Typischer Funktionsumfang:

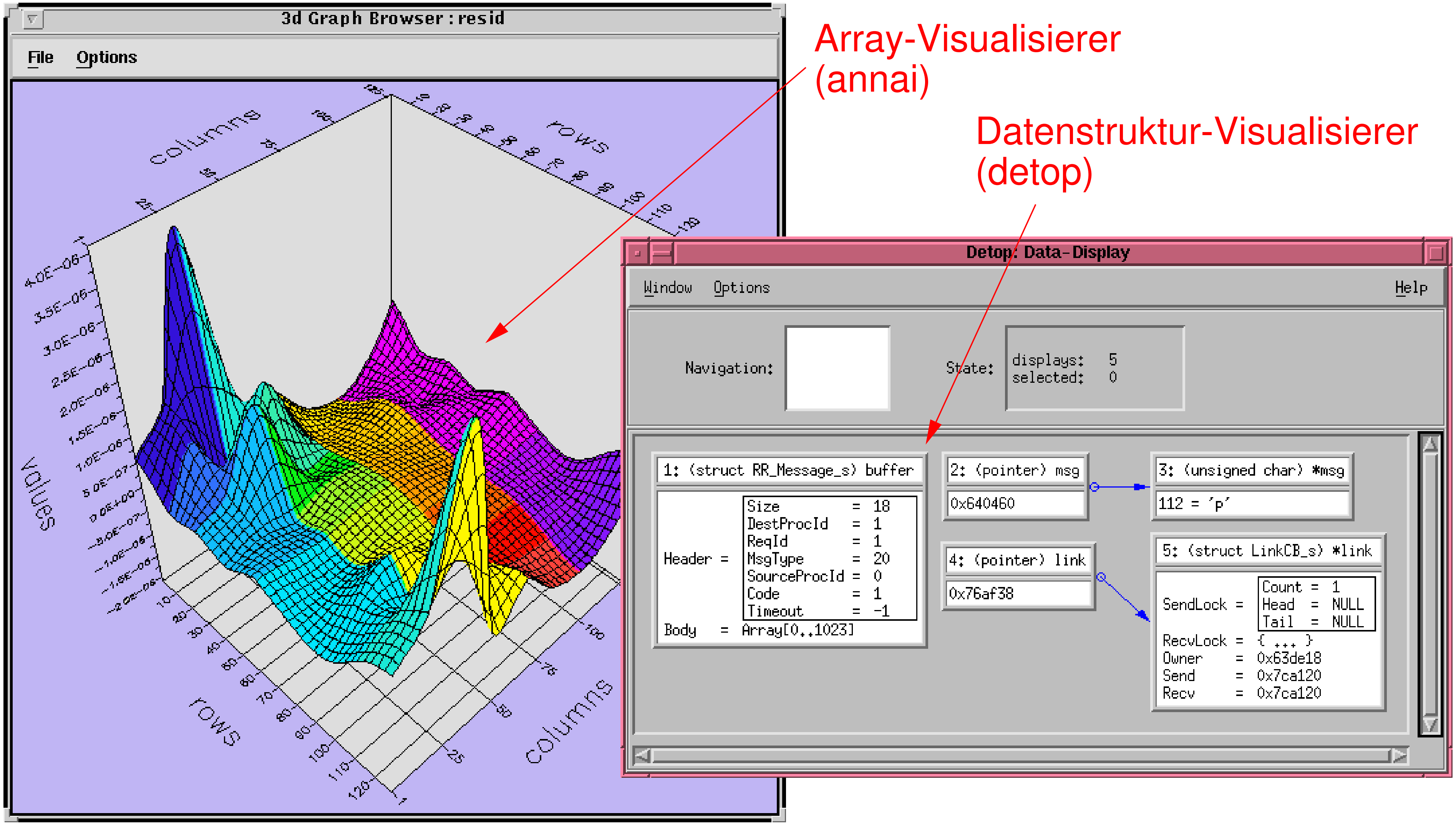
- Modifikation des Programmzustands
  - Setzen von Variablenwerten, Verändern des Kontrollflusses $\leadsto$ Testunterstützung
    ```
    set i = 17
    set x = foo(ptr,"hallo")
    jump 113
    ```
  - Selten: *code patching*
- Ausführungskontrolle
  - Starten und Stoppen des Prozesses
    Start des Programms: `run -o tst`
    Stoppen und fortsetzen: `^C` und `continue`
  - Einzelschritt-Abarbeitung
    Varianten: `step` (in Funktionen hinein) und `next` (Funktionsaufrufe überspringen)

Auch Kombinationen und Abwandlungen: Variablen-Display, Visualisierer, *tracepoints*, ...

Beispiele für Datenvisualisierung

Array-Visualisierer
(annai)

Datenstruktur-Visualisierer
(detop)



Konzepte paralleler Debugger

Eigenschaften paralleler Programme

- Mehrere Aktivitätsträger
  Prozesse, evtl. auch Threads (auch heterogen)
- Dynamik
  Erzeugen und Löschen von Prozessen, Threads, Kommunikationsverbindungen etc. zur Laufzeit
- Interaktion
  Kommunikation und Synchronisation zw. Prozessen bzw. Threads
- Verteiltheit
  Verteilte Information, kein einfach zu bestimmender globaler Systemzustand

$\leadsto$ Berücksichtigung dieser Eigenschaften in existierenden Debuggern sehr unterschiedlich gelöst

$\leadsto$ Im Folgenden nur Beispiele, es gibt keine Standards

Umgang mit mehreren Prozessen

Zwei Methoden: **Fenstertechnik und Prozeßmengen**

Varianten:

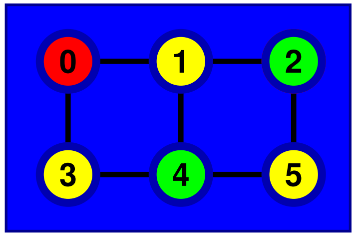
- Pro Prozeß ein Fenster
  - Unabhängiger, sequentieller Debugger in jedem Fenster (z.B. Debugging-Unterstützung in PVM)
  - Paralleler Debugger (z.B. `totalview`)
    $\leadsto$ (v.a.) geeignet für funktionsparallele Programme
- Ein einziges Fenster für alle Prozesse
  - Auswahl genau eines Prozesses zum Debugging (z.B. `p2d2`)
  - Kommandos für Prozeßmengen (z.B. `prism`, `(x)ipd`)
    $\leadsto$ (v.a.) geeignet für datenparallele Programme
- Mehrere Fenster für beliebige Teilmengen der Prozesse (z.B. `detop`)
  $\leadsto$ für funktions- und datenparallele Programme

Prozeßauswahl

Auswahl über Prozeßnummer und Knotennamen/-nummer sehr unkomfortabel

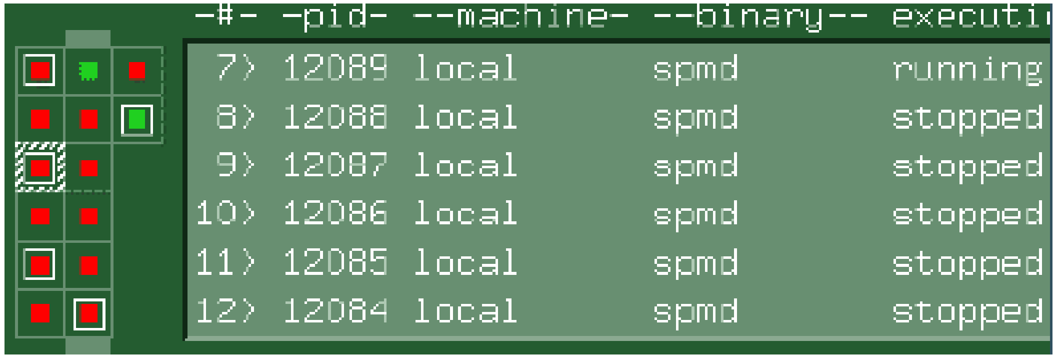
Unterstützung der Prozeßauswahl:

- Graphisch über Topologie
  (statisch, oft 1 Prozeß/Knoten)
  Beispiel: `xipd`



- Textuell mit Zusatzinformation
  Beispiel: `detop`

- Graphisch und textuell
  Beispiel: `p2d2`



- Quasi-Standard: Zusatzinformation beinhaltet Prozeßzustand

Skalierbarkeit und Dynamik

Problem: Umgang mit höheren Prozeßanzahlen

Lösungen:

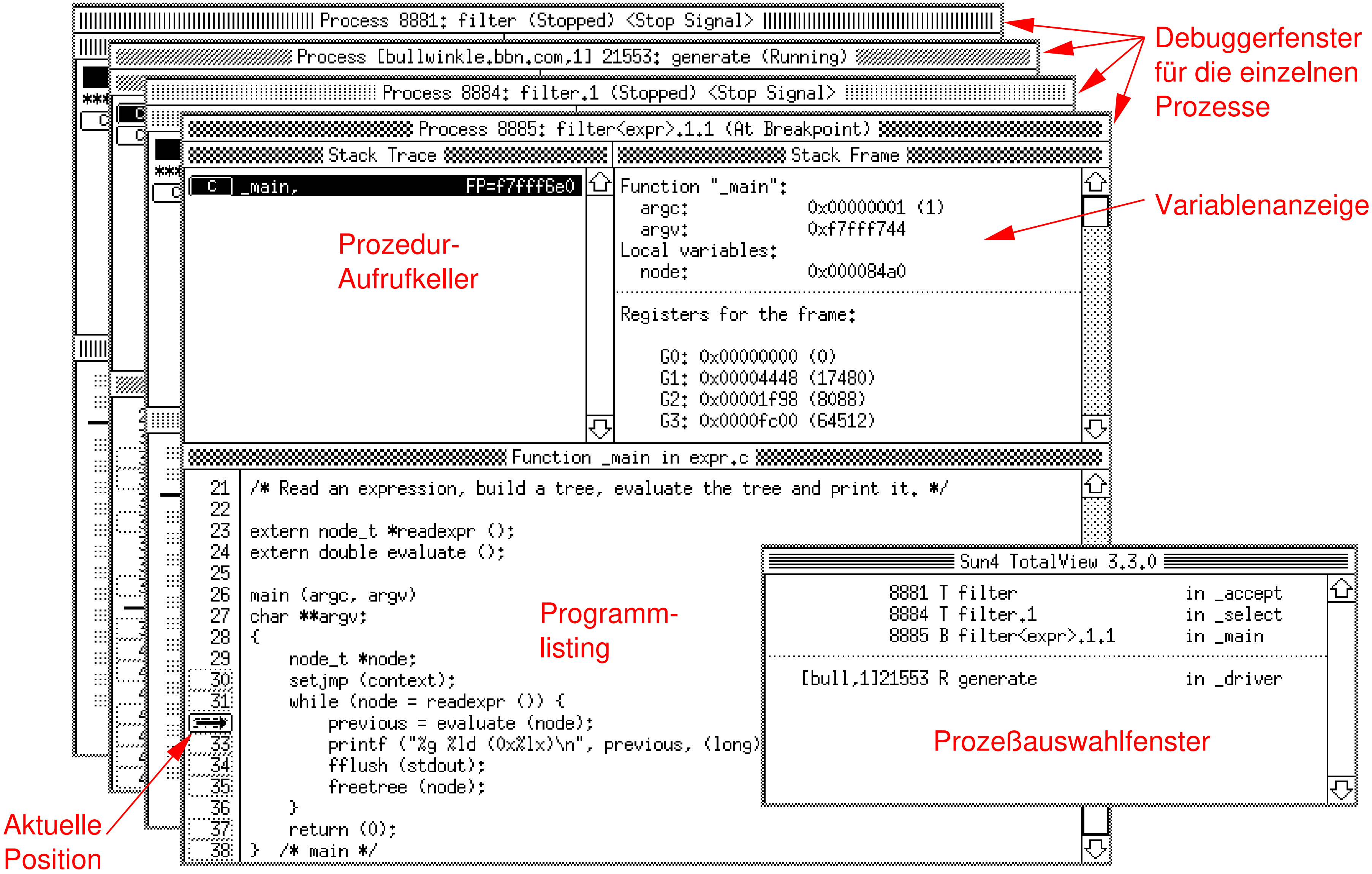
- Kommandos für Gruppen von Prozessen
- Zusammenfassen identischer Ergebnisse für verschiedene Prozesse
- Einsatz von graphischen Darstellungen für Zustandsanzeigen
  - Benutzung von Farbe oder Icons in Prozeßlisten
  - Spezielle Visualisierer (z.B. Aufrufbaum)

Problem: Debugging dynamisch erzeugter Prozesse

Lösungen:

- Stoppen aller neu erzeugten Prozesse, manuelle Auswahl
- Automatische Auswahl über Mustervergleich mit Zusatzinformation

Beispiel: `totalview`



Beispiel: p2d2

Prozeßstatus und -auswahl

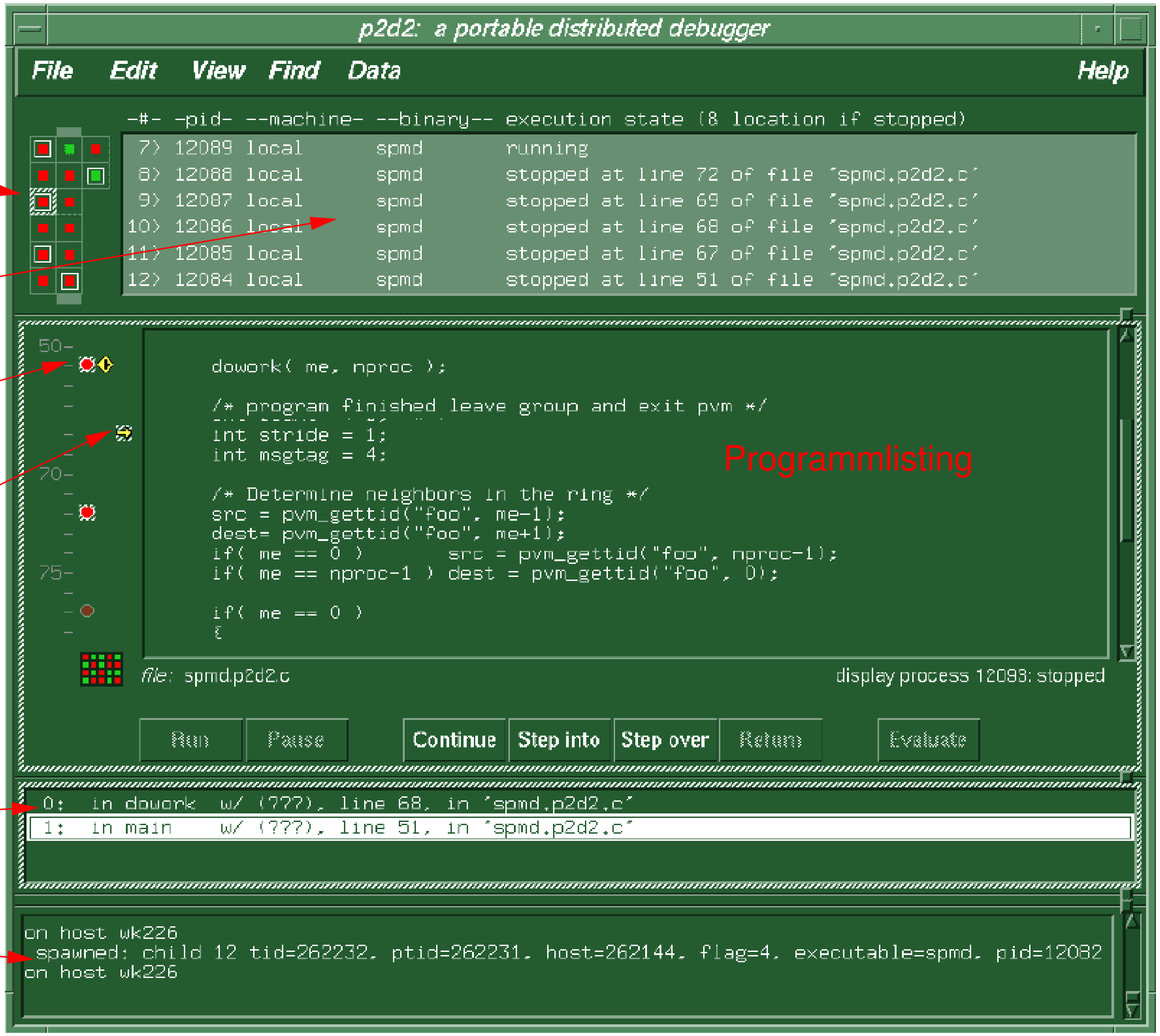
Zustand der ausgewählten Spalte

Haltepunkt

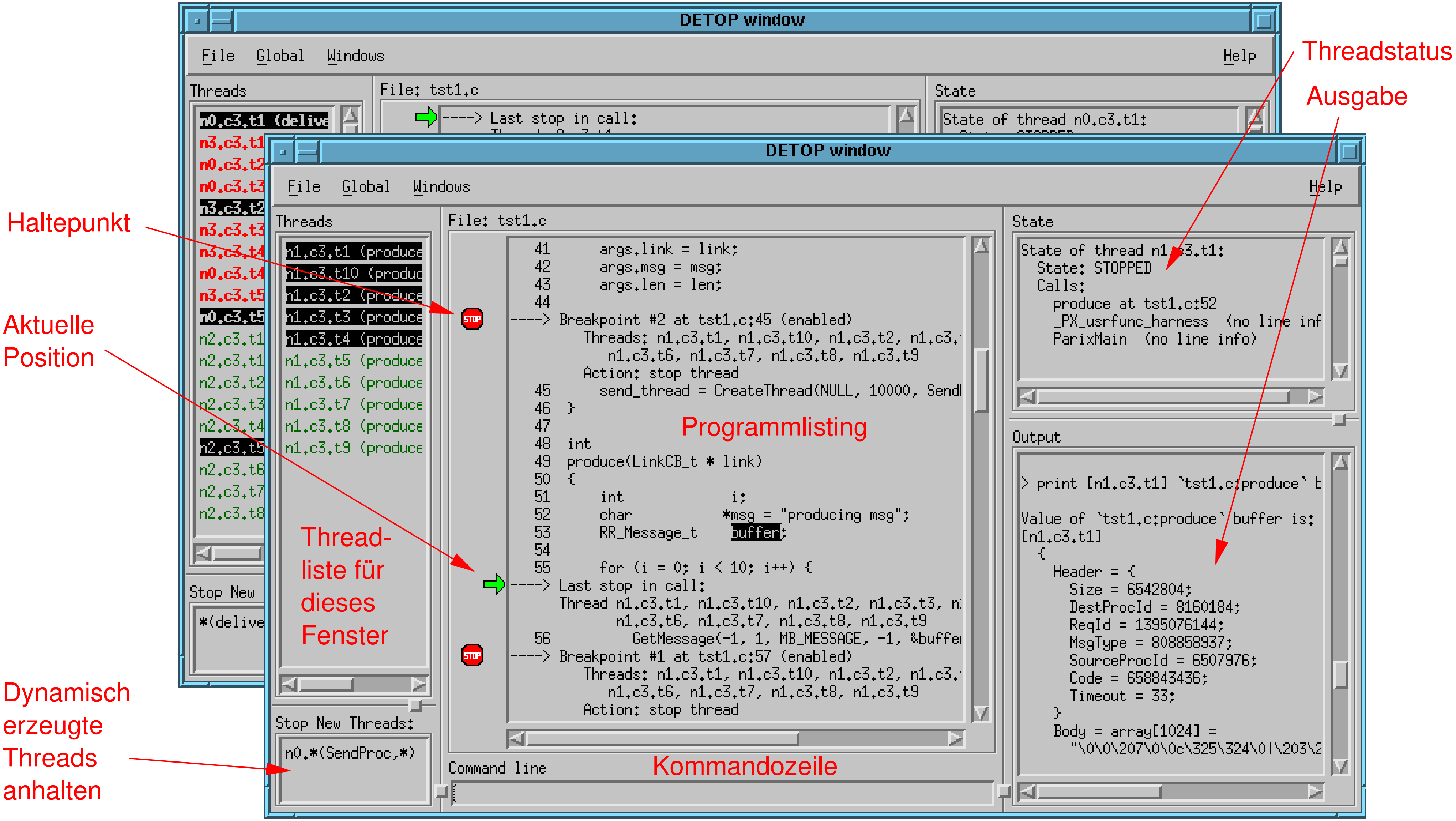
Aktuelle Position

Prozedur-Aufrufkeller

Ausgabe



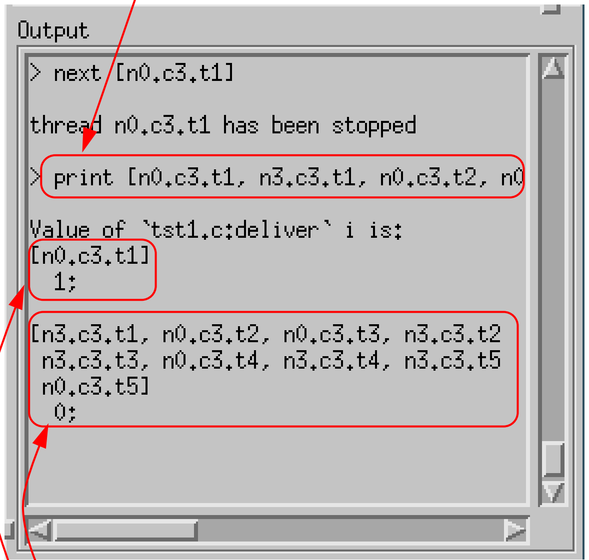
Beispiel: `detop`



Beispiel: `detop`

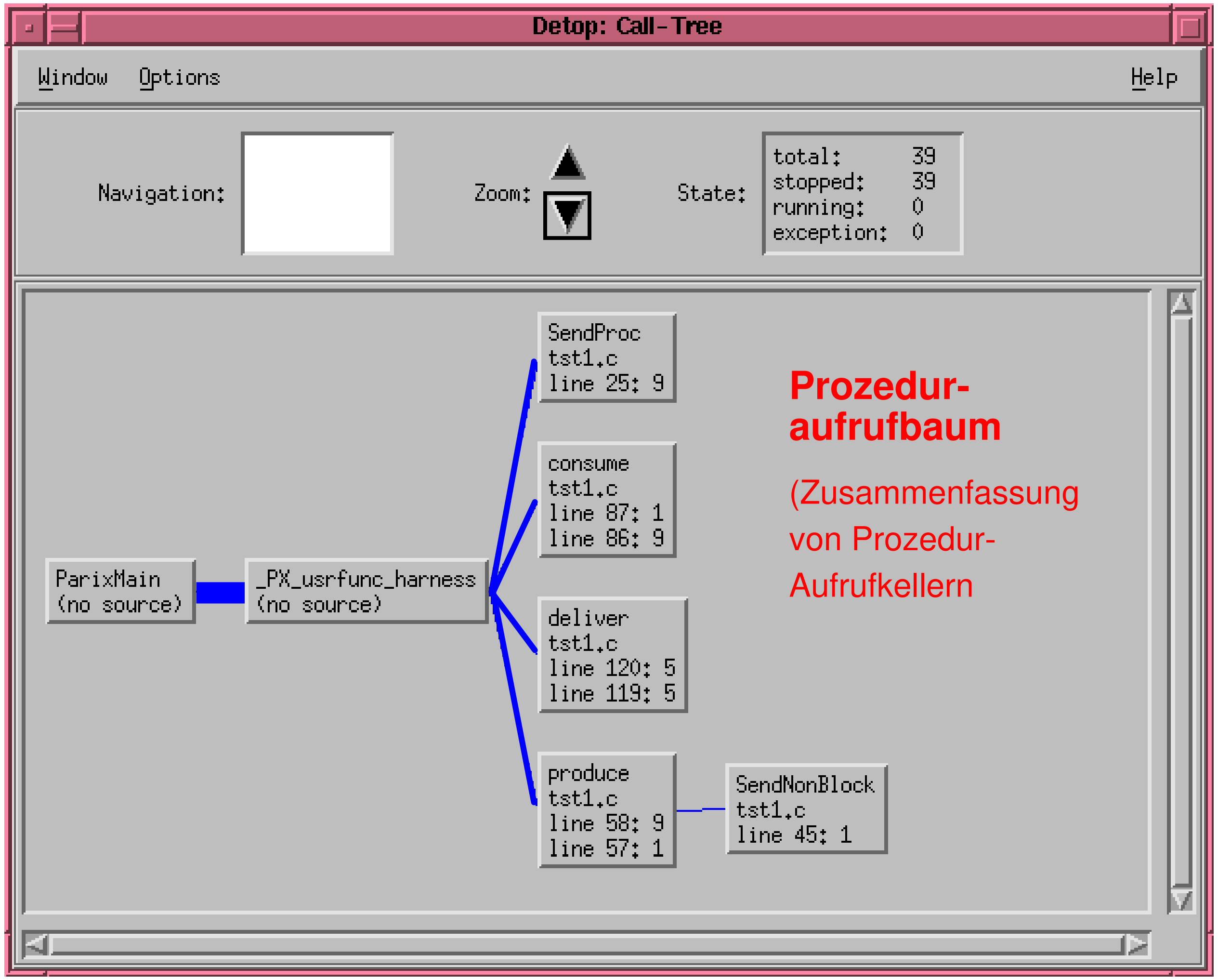
Ergebniszusammenfassung

Wert von i in 10 Threads



9 Threads: Wert 0

1 Thread: Wert 1



Prozedur-aufrufbaum

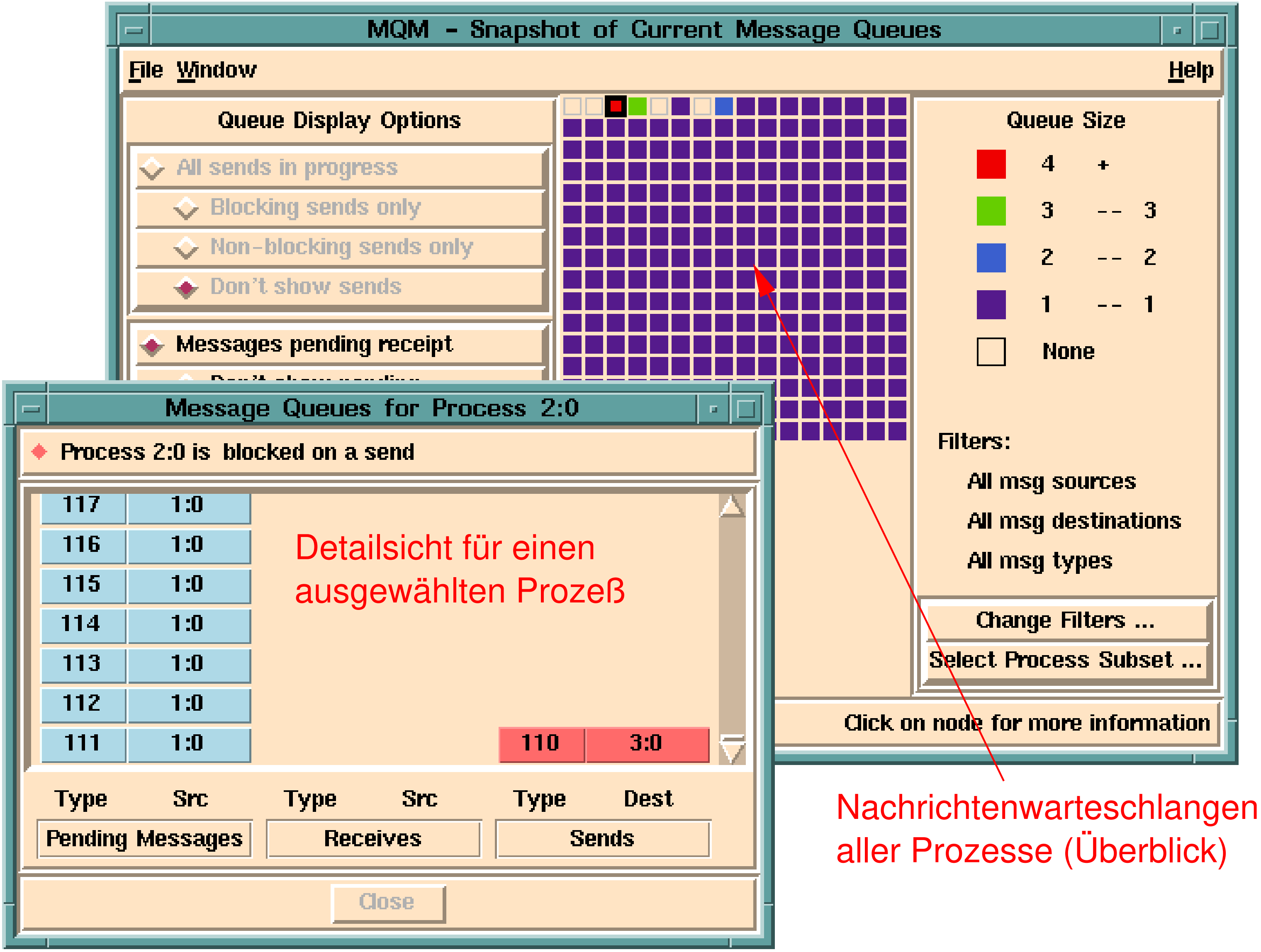
(Zusammenfassung von Prozedur-Aufrufkellern

Interaktion

Überwachung von Kommunikation und Synchronisation

- Möglich durch (evtl. auch bedingte) Haltepunkte auf entsprechende Bibliotheksfunktionen
- Meist keine weitergehende Unterstützung, etwa:
  - Ausgabe wartender Prozesse
  - Status von Nachrichtenwarteschlangen oder Kommunikationsverbindungen
  - Haltepunkte auf Nachrichten („wie wird die Nachricht beim Empfänger weiterverarbeitet?“)
- Auswege:
  - Gleichzeitige Benutzung spur-basierter Werkzeuge (aber: i.a. keine Integration)
  - Spezielle Werkzeuge, etwa `mqm`

Beispiel: `mqm`



Verteiltheit

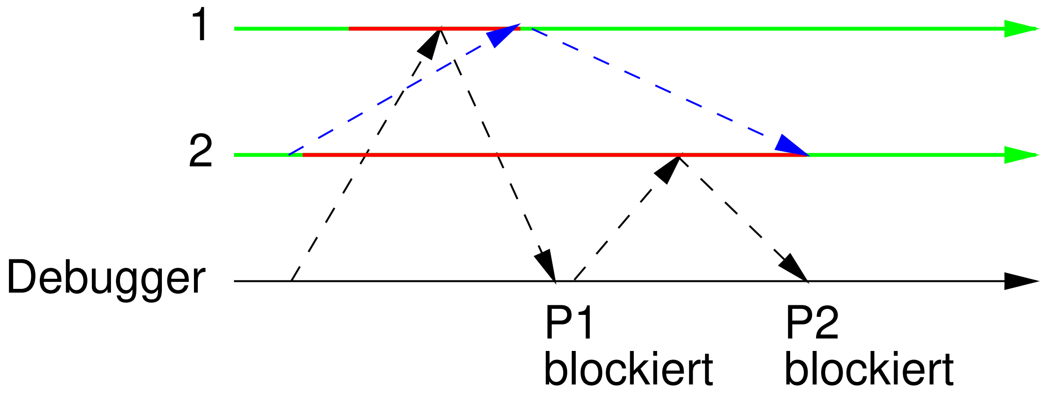
Unterschiedliche Aspekte:

- Verteilte Daten der Anwendung
  $\rightsquigarrow$ Visualisierer für verteilte Arrays, z.B. `DAQV` (PTOOLS-Projekt)
- Globale Daten verteilter Prozesse
  - Beispiele: PVM-Gruppen, gemeinsame Speichersegmente
  - Problem: Einfrieren des Zustands bei Erreichen eines Haltepunkts
  - Meist nur Anhalten des Prozesses möglich, der Haltepunkt erreicht hat
    $\Rightarrow$ andere Prozesse können Zustand verändern
  - Globales Anhalten i.a. nicht instantan möglich
    $\Rightarrow$ Zustand nach Anhalten nicht garantiert identisch mit Zustand am Haltepunkt
- Globale Ereigniserkennung (Forschungsthema!):
  Verknüpfung von Ereignissen in verschiedenen Prozessen
  z.B. a tritt (kausal) nach b auf; a und b treten kausal ungeordnet („gleichzeitig“) auf

Beispielszenarien

Deadlocks

- Im Prinzip einfach: Abfrage des Status aller Prozesse
  Im Deadlock-Fall sind alle Prozesse blockiert
- Aber:
  - Erkennung der Blockierung z.T. nur durch Inspizieren des Aufrufkellers (nicht eindeutig!)
  - Für konsistenten Globalzustand: alle Prozesse müssen gestoppt werden!



Beispielszenarien (2)

Races

- Zur reinen Erkennung von Races: Haltepunkt-basierte Debugger wenig geeignet
  Besser:
  - Laufzeit-Debugger
  - Spur-basierte Werkzeuge (Vergleich zweier Spuren, Erkennung jedoch Zufall!)
- Problem bei Vorhandensein von Races:
  Fehler bei Einsatz eines Debuggers evtl. nicht reproduzierbar.
- Mit genauem Wissen über die Stelle des Races ist jedoch das Ergebnis des Races i.a. über Anhalten und Fortsetzen einzelner Prozesse steuerbar.
- Zur Steuerung besser geeignet:
  Werkzeuge zur deterministischen Ausführung (Forschungsthema!)
  - replay-Techniken
    z.B. in Werkzeugumgebung `annai` (wird Produkt für SX-4)
  - Steuerung über Zugriffsmuster für gemeinsame Objekte

Vorbereitung und Bewertung

Vorbereitung der Anwendung:

- Übersetzung mit spezieller Compileroption ('`-g`')
  - Eincodieren von Symbolinformation in die Binärdatei
  - Meist auch Abschalten aller Compileroptimierungen
  - Optimierungen führen zu „merkwürdigem“ Verhalten des Debuggers
    $\Rightarrow$ vorsichtshalber Optimierungen explizit abschalten ('`-O0`')

Bewertung:

+ Haltepunkt-basierte Debugger vor allem geeignet für detaillierte Fehlersuche
− Zur Zeit i.w. nur Unterstützung für „sequentielle“ Fehler
+ Trotzdem das effizienteste Werkzeug zum Lokalisieren von Fehlerursachen