ÜBERBLICK

- ❏ Parallelrechnerarchitekturen und Programmiermodelle
- ❏ HP - PA 7100
- ❏ Convex SPP Architektur
- ❏ Parallele Anwendungen
- ❏ Systemzugang und Besonderheiten
- ❏ Convex SPP C Compiler
- ❏ Automatische Parallelisierung
- ❏ Convex SPP Speicherklassen
- ❏ Parallele Programmierung mit Threads (CPS-Library)
- ❏ Parallele Programmierung mit Threads (Thread System Calls)

Programmiermodelle

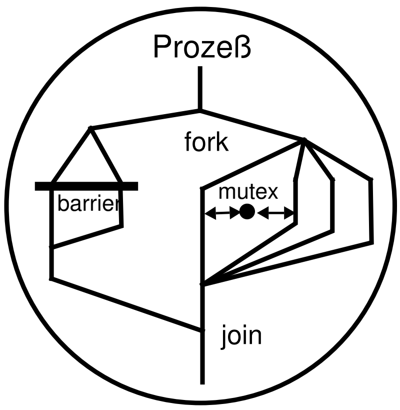
Parallele Prozesse + Message Passing

send(Botschaft, Empfänger-Adresse)
Botschaft = receive(Sender-Adresse)

Parallele Prozesse + Shared Memory

shm_id = shm_get(Schlüssel, Größe, Zugriffsrechte)	
ptr = shm_at(shm_id)	shm_dt(prt)
semaphore = sema_create(Schlüssel, Zugriffsrechte)	
wait(semaphore)	signal(semaphore)

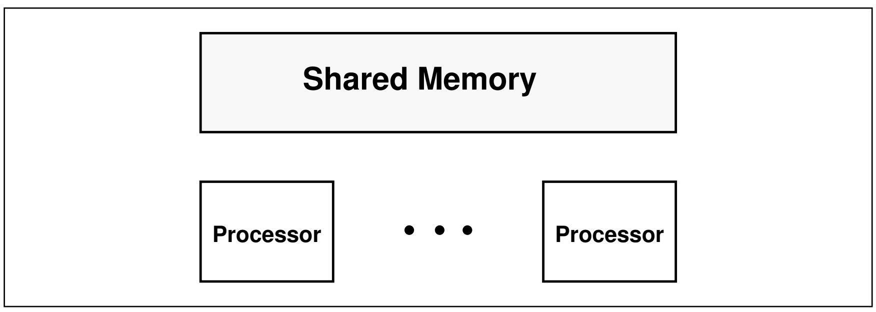
1 Prozeß + Parallele Threads + Shared Memory



Programmiermodelle

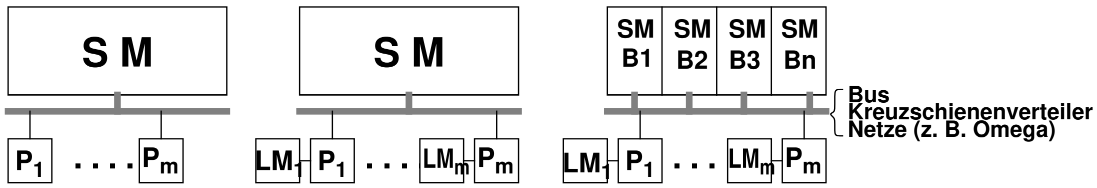
- ❏ Abbildung auf Hardwarearchitektur
  - ❏ Message Passing Architekturen
    (Ncube, IBM SP2, ...)
    - Parallele Prozesse + Message Passing
    - Parallele Prozesse + Shared Memory (sehr eingeschränkt)
  - ❏ Shared Memory Architekturen
    (Convex SPP (NUMA), KSR (NUMA), SGI PowerChallenge (UMA), ...)
    - Parallele Prozesse + Shared Memory
    - Parallele Threads + Shared Memory
    - Parallele Prozesse + Message Passing

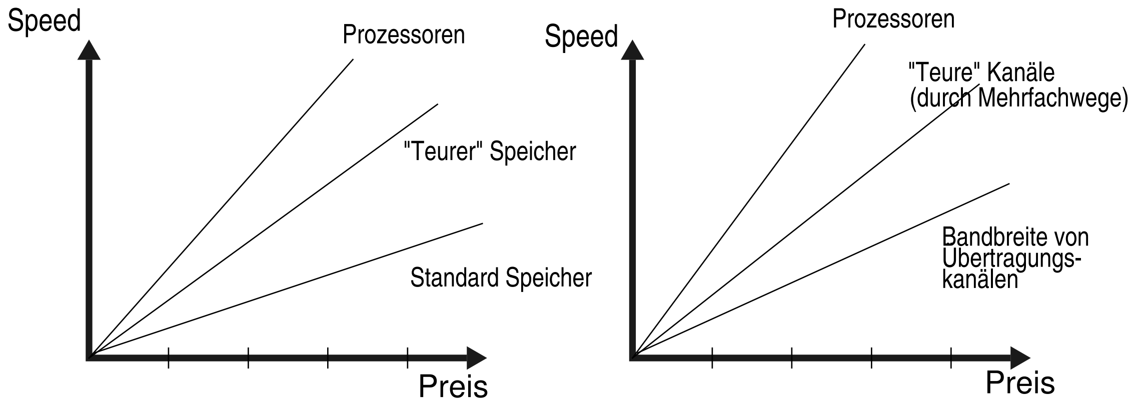
Ideales Shared Memory System



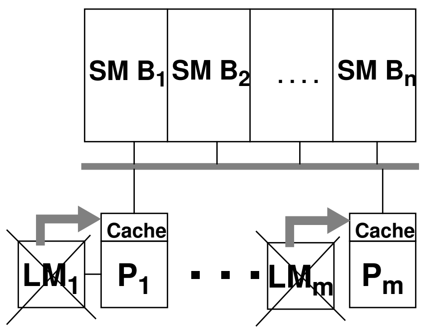
- ❏ Viele Prozessoren greifen uniform auf einen gemeinsamen Speicher zu
- ❏ Die Architektur ist symmetrisch
- ❏ Alle Komponenten sind gleichberechtigt (z.B. bei I/O)

Entwicklung der Shared Memory Systeme

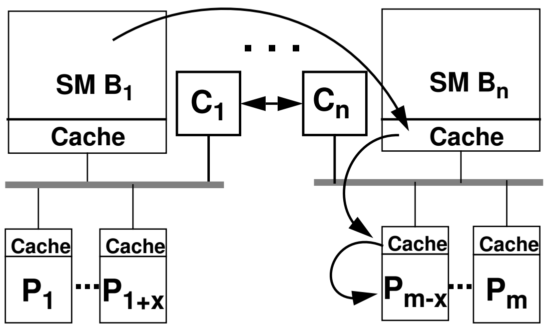




Entwicklung der Shared Memory Systeme



z.B. Sequent ; Encore (UMA)



z.B. Convex SPP (NUMA)

- ❑ Übertragungsbandbreite:
  - ❑ "elektrische" Probleme
  - ❑ Protokoll-Probleme
  - ❑ Entfernungsproblem (Lichtgeschwindigkeit: 30cm/ns)
    - Packungsdichte
- ❑ Ausgewogenheit:
  - ❑ Prozessorleistung
  - ❑ Speicher(abschnitts)größen
  - ❑ Übertragungsbandbreiten
  - ❑ Cachegrößen

Merkmale von UMA und NUMA Architekturen

	UMA	NUMA
Speicherzugriff	uniform	nicht uniform
Caches	Prozessorcaches (intern und/oder extern)	oft Cache-Hierarchien
Prozessoranzahl	in der Regel < 32	> 100 möglich
Verbindungs-netzwerk	Bus oder Crossbar	Hochgeschwindigkeits-netzwerk (event. in Hierar-chien oder in Kombination mit Bus oder Crossbar)

Das Programmiermodell Threads

- ❏ Ein Prozeß kann aus mehreren Threads bestehen
  - ❏ Der Prozeß bildet die Umgebung (Adreßraum, Schutzmechanismen, Filedeskriptoren, ...)
  - ❏ Die Threads entsprechen unabhängigen Ablaufpfaden
  - ❏ Jeder Thread verfügt über einen eigenen Stack, eine Registerbelegung und einen Programmzähler (private Variablen, unabhängiger Programmablauf)
  - ❏ Ein “Unix Prozeß” entspricht einem Prozeß mit einem Thread
- ❏ Auf einem Multiprozessor können Threads eines Prozesses parallel von mehreren Prozessoren abgearbeitet werden
- ❏ Threads können zwischen Prozessoren wandern
- ❏ Durch die gemeinsame Umgebung ist eine schnellere Umschaltung zwischen Threads eines Prozesse möglich (im Vergleich zu einer Prozeßumschaltung)

Das Programmiermodell Threads

- ❑ Elemente des Programmiermodells
  - ❑ Thread-Identifikatoren zur Erzeugung, Terminierung, Suspendierung und Aktivierung der Threads
  - ❑ Synchronisationsvariablen für gegenseitigen Ausschluß (mutex) (Beispiele im Programmierteil)
  - ❑ Barrieren (Beispiele im Programmierteil)
  - ❑ Weitergehende Synchronisationsmechanismen (auf SPP nicht verfügbar)
    - Bedingungsvariablen
    - Monitore
- ❑ Gemeinsamer Speicher ist der "Normalfall"
  - ❑ Threads eines Prozesses laufen im gleichen Adressraum
  - ❑ Bei einer Programmierung auf Basis von Prozessen müssen gemeinsame Speicherbereiche explizit angefordert werden

Das Speichermodell

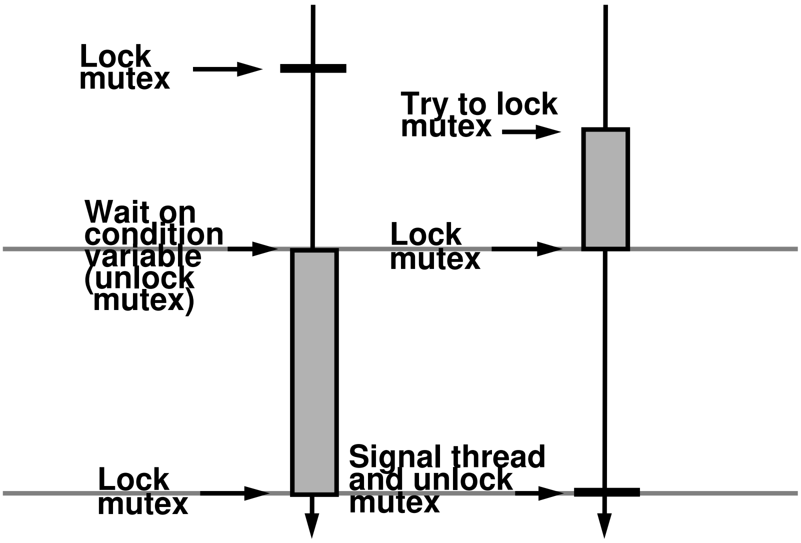
- ❑ Der Programmierer ist verantwortlich für:
  - ❑ den gegenseitigen Ausschluß beim Schreiben in den gemeinsamen Speicher
  - ❑ die Deklaration, welche Daten für alle Threads und welche nur privat für einen Thread zugänglich sind

  --> Datenkonsistenz
- ❑ Die Rechnerarchitektur garantiert lediglich:
  - ❑ nach der Zuweisung eines neuen Wertes an eine gemeinsame Variable liefert (auf allen Prozessoren) die Ausgabe des Variablenwertes diesen neuen Wert zurück
  - ❑ Lock- und Unlock-Operationen auf Synchronisationsvariablen sind zeitlich geordnete Ereignisse (atomare Speicheroperationen)

  --> Cache- und Speicherkonsistenz

Bedingungsvariablen

❏ Bedingungsvariablen zum bedingten Eintritt in einen kritischen Abschnitt



Vergleich: Threads - Message Passing (PVM)

Threads

```
main()
{
   for (all threads)
      thread_create();

   work();

   for (all threads)
      thread_join();
}
```

PVM-Master

```
main()
{
   pvm_mytid();
   pvm_spawn("slave",
                  slaves)

   pvm_initsend();
   pvm_pk(feld);
   pvm_mcast(slaves);

   work();

   for (all slaves) {
      pvm_recv();
      pvm_upk();
   }
   pvm_exit()
}
```

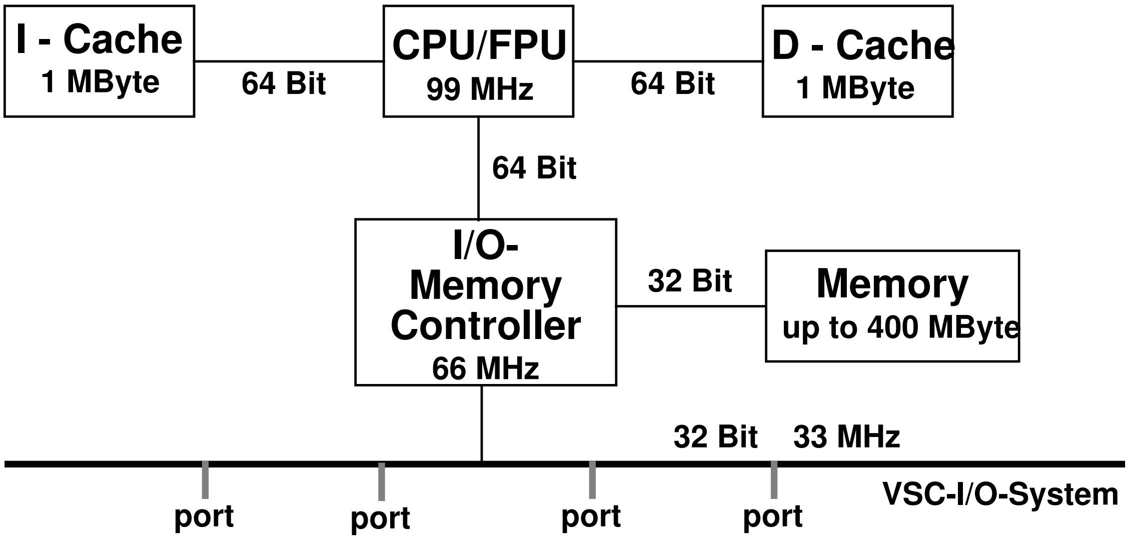
PVM-Slave

```
main()
{
   pvm_mytid();
   master_id=pvm_parent()
   pvm_recv();
   pvm_upk(feld);

   work(feld);

   pvm_initsend();
   pvm_pk();
   pvm_send(master_id);
   pvm_exit()
}
```

HP - PA 7100: Überblick



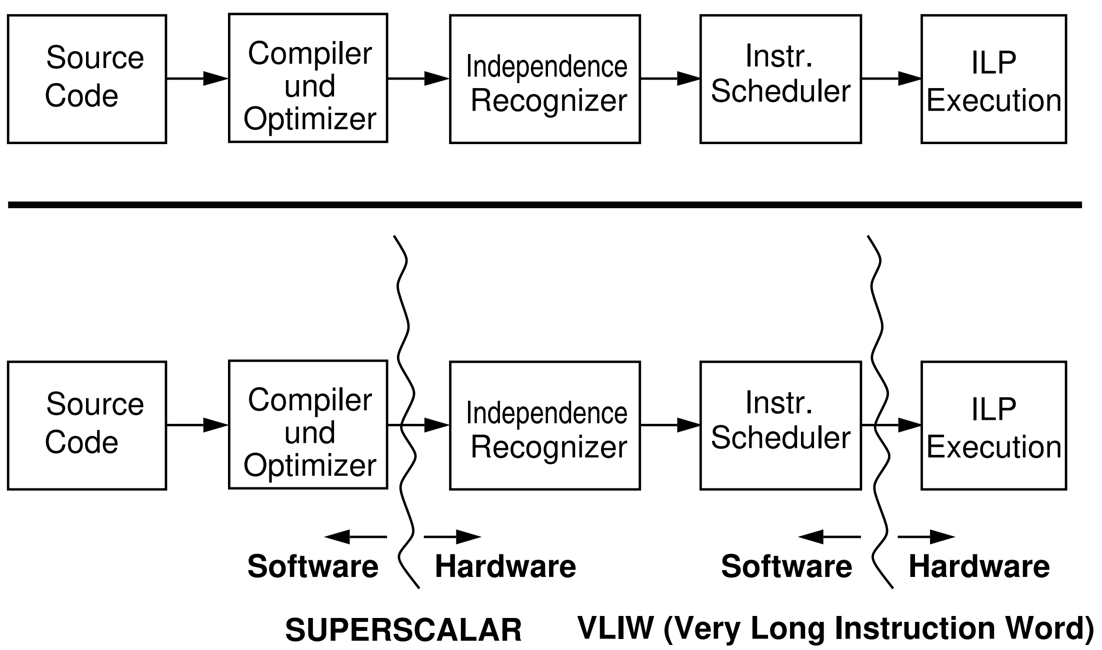
HP - PA 7100: Überblick

- ❏ CPU:
  - ❏ 100 MHz; CMOS, 0.8 micron
  - ❏ 32 x 32 Bit-General-Register
  - ❏ 5 Stage Pipelines
- ❏ Datenformate:
  - ❏ Signed & Unsigned 16-bit-Integers
  - ❏ Signed & Unsigned 32-bit-Integers
  - ❏ Single Word (32-bit) IEEE Floating Point
  - ❏ Double Word (64-bit) IEEE Floating Point
  - ❏ Quad Word (128-bit) IEEE Floating Point

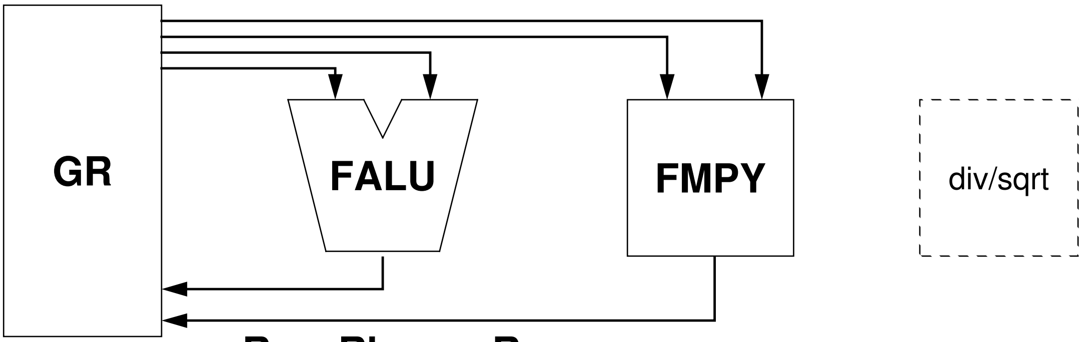
Parallelität auf Instruktionsebene

- RISC (Reduced Instruction Set Computing)
  - (Ur)-Ziel: 1 Takt; 1 Befehl
- Komplexere Befehle werden auf einfache Befehle zurückgeführt oder spezielle “Functionale Units” (CISC ?)
- Überlappende (parallele) Ausführung durch Einführung von Pipelining
  - Probleme: Datenunhängigkeit; Branching
- Mehrere “Functionale Units” parallel
  - Probleme: Datenabhängigkeit

Parallelität auf Instruktionsebene: HW / SW



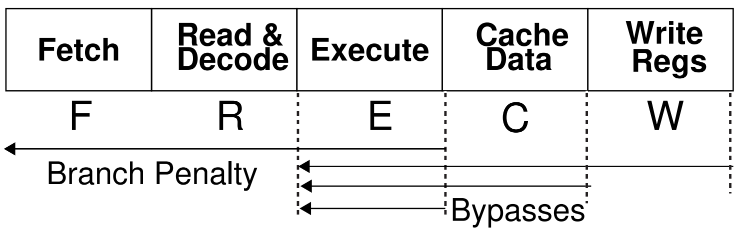
HP - PA 7100: Innerer Aufbau

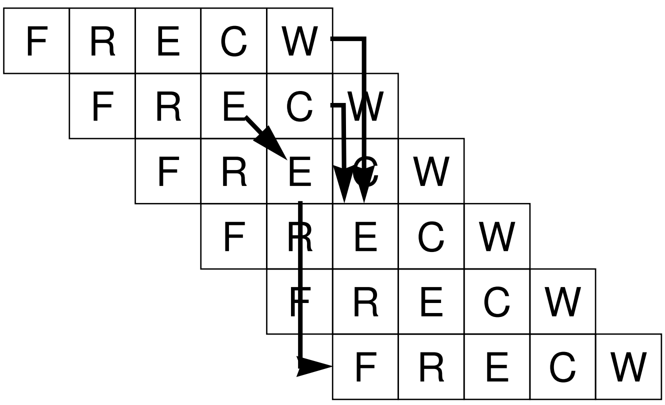


$$Ra \times Rb \longrightarrow Rm$$
$$Rs + Rc \longrightarrow Rs$$

- ❏ 3 unabhängige Funktional Units
  - Floating Point Alu
  - Floating Point Mult
  - Floating Point Div/Sqrt
- ❏ Mult/Add gleichzeitig möglich
  - 198 MFlops Peak
  - Kein Code gefunden!

HP - PA 7100: Innerer Aufbau (Path-Length-Reduction)

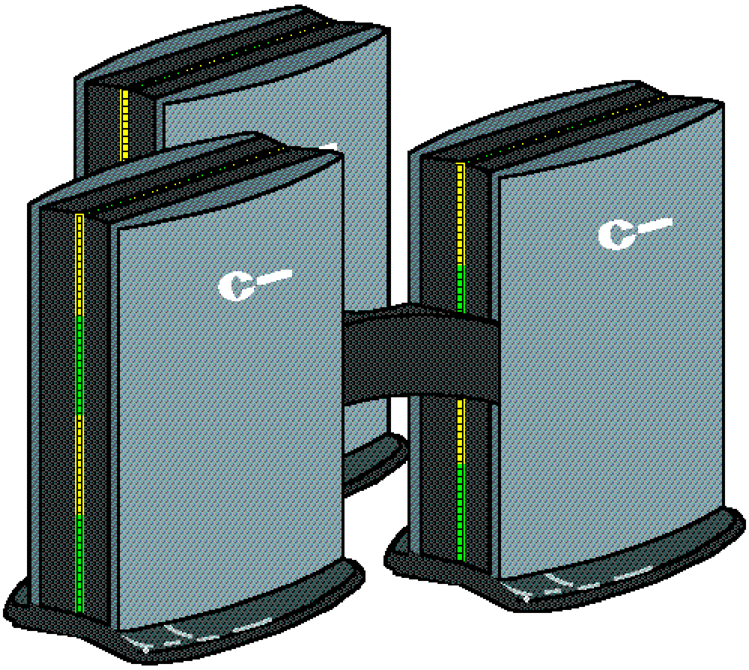




HP - PA 7100: Leistungsdaten

- DATA Cache: 1 MByte
  - Read: 1 Cycle / 8 Byte --> 792 Mbyte / s
  - Write: 2 Cycle / 8 Byte --> 396 Mbyte / s
- Instr. Cache: 1 MByte
  - Read: 1 Cycle / 8 Byte--> 792 MByte / s
- Memory System: siehe SPP Architektur

Convex SPP am RRZE



- 6 Hypernodes mit je 8 Prozessoren (= 48 Proz.)
- 6 Gigabyte Hauptspeicher
- 46 Gigabyte Plattenspeicher
- DAT-Tape
- FDDI
- Teststation

Convex SPP Architektur

- ❏ Prozessor: HPPA 7100 / 100 MHz
- ❏ Prozessor-Cache: 1 MB Daten / 1 MB Instruktionen
- ❏ Verbindungshardware:
  - ❏ 8 Prozessoren über non blocking Crossbarswitch (Hypernode)
  - ❏ max. 16 Hypernodes über 4 parallele Hochgeschwindigkeitsringe
  - ❏ --> Non Uniform Memory Access (NUMA)
  - ❏ Network-Cache
- ❏ I/O-System
  - ❏ 250 MB/sec I/O Port an jedem Crossbar (also auf jedem Node)
  - ❏ 4 (8) x SBus (Sun) pro I/O Port
    - SSCI-2 F&W, FDDI, ...

Convex SPP Architektur

- ❏ Betriebssystem:
  - ❏ SPP-UX (MACH 3.0 Microkernel, HP-UX Server)
    - Multiple Servers
  - ❏ HP-UX Application Binary Interface (ABI)
    - HP-UX Programme sind i. a. ausführbar
  - ❏ Affinity Scheduler
    - Scheduler versucht Prozesse an ihren Prozessoren zu halten
    - Wichtig für Cacheausnutzung
    - Insbesondere auf NUMA-Architekturen

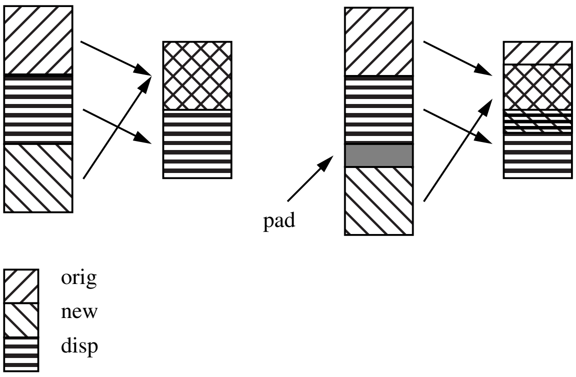
Speicher-Architektur / Cache

- Prozessor-Cache
  - 1 MB Datencache / 1 MB Instruktionscache
  - 32 Byte Cacheline
  - Direct mapped / Virtually indexed / Write back
  - Abbildung: Speicher -> Cache

    cache_address = MOD(virtual_adress, 2^{20})

- Network-Cache
  - Größe konfigurierbar
  - 64 Byte Cacheline
  - Physically indexed

Cache trashing (single processor)



- Beispielprogramm

```
int orig[1024*128];   /* 512 KB */
int new[1024*128];    /* 512 KB */
int pad[16];          /* 64 byte - 2 c-lines */
int disp[1024*128];   /* 512 KB */

for (i=0, i<=1024*128, i++)
{
        new[i] = orig[i] + disp[i];
}
```

Speicher-Architektur / Speicherklassen

- ❏ Physikalische Speicherklassen
  - ❏ Node private
    - abh. vom Speicherausbau / 50 cycles
  - ❏ Prozessor Cache
    - 2 x 1 MByte / 1 cycle
  - ❏ Subcomplex global
    - abh. vom Speicherausbau / 50 cycle oder 200 cycles
  - ❏ Network cache
    - konfigurierbar / 50 cycle

Speicher-Architektur / Speicherklassen

- ❏ Virtuelle Speicherklassen
  - ❏ Thread private
  - ❏ Node private
  - ❏ Near shared
  - ❏ Far shared
  - ❏ Block shared

Literatur

- ❏ Rechnerarchitektur von RISC Prozessoren:
  Hennessy, Patterson, “Computer architectur, a quantitative approach”,1990, 14GI/mat 10.2-347
- ❏ Rechnerarchitektur von Parallelrechnern:
  Hwang, “Advanced computer architecture”, 1993, 14GI/mat 10.2-427
- ❏ Wegweisende NUMA-Rechnerarchitektur:
  J. Hennessy, “The Stanford DASH Multiprocessor”, IEEE Computer, 3/1992, zu finden in der GI bei den gebundenen Zeitschriften
- ❏ Convex Exemplar Architecture Guide
  /local/doc/spp/ps/arch.ps
- ❏ Convex Exemplar Programming Guide
  /local/doc/spp/ps/EPG.ps

Parallele Anwendungen

- ❏ Shared Memory (CPS-Library (**C**ompiler **P**arallel **S**upport))
  - ❏ Automatische Parallelisierung durch den Compiler (Schleifen)
  - ❏ Parallelisierung durch den Compiler über Direktiven (Schleifen, ...)
  - ❏ Parallelisierung per Hand
  - ❏ Ein Thread pro Prozessor (z.Z. kein „oversubscription”)
  - ❏ Automatische Anpassung an verfügbare Prozessoranzahl (kein neues Übersetzen oder Linken notwendig)
  - ❏ Alle Threads teilen einen Adreßraum (private Daten möglich)
  - ❏ Nur innerhalb eines Subcomplexes
  - ❏ aber auch:
    - Parallelisierung auf niedrigerer Ebene (Systemfunktionen)
    - Mehr Threads als Prozessoren (auf BS-Ebene! / Effizienz?)
    - Weniger komfortabel

Parallele Anwendungen

- ❑ Message Passing
  - ❑ Mehrere unabhängige Prozesse
    - • Innerhalb eines Subcomplexes (Speicherkopplung)
    - • Mehrere Subcomplexe und externe Maschinen (Sockets)
  - ❑ Programmiermodell: PVM (3.2.6)
  - ❑ Kommunikation über Nachrichten
  - ❑ Gemeinsame Speicherbereiche nur über spezielle Segmente (memory mapped files)
- ❑ Hybride Applikationen
  - ❑ Ein PVM Prozeß kann multithreaded sein
  - ❑ z.B. PVM-Mechanismen zwischen Subcomplexen

Convex SPP - Systemzugang

- ❑ Interaktiver Zugang
  - ❑ gsm (131.188.3.30)
  - ❑ Über **`rlogin`** oder **`telnet`**
- ❑ Konfiguration (Subcomplexe)
  - ❑ System (nur für Systemdienste, 2 Prozessoren)
  - ❑ **default** (6 Prozessoren, 256 MB Global Memory)
  - ❑ gsm8 (8 Prozessoren, 512 MB Global Memory)
  - ❑ gsm32 (32 Prozessoren, 4 Nodes, 4 * 512 MB Global Memory)
- ❑ Batch Zugang
  - ❑ Über NQS (**`qsub, qstat, qjob, qdel`** - man pages)
  - ❑ Eingangsqueues: **`gsm8@gsm`** oder **`gsm32@gsm`**
  - ❑ **Nicht für Übungsaufgaben (keine lange Laufzeit)!!!!!**

Convex SPP - Besonderheiten

- Allgemeines
  - /usr/convex/man in MANPATH aufnehmen
  - fau_top - wie top, gibt zusätzlich den Subcomplex
- Subcomplexe
  - Nach Login automatisch in default
  - Starten von Prozessen in anderen Subcomplexen über mpa
    - **`mpa -sc gsm32 par_test`** startet par_test in gsm32
    - **`mpa -sc gsm8 csh`** startet eine C-Shell in gsm8 (inkl. aller Child-Prozesse)

Convex SPP - Besonderheiten

- Parallele Programme
  - **`cnx_ps -T`** zeigt einzelne Threads eines Prozesses an (versteht alle Optionen von ps)
  - pot - ein top auf Basis von Phreads (siehe man page)
  - Zeitmessung über Kommandozeile
    - **`timex`** oder **`/bin/time`** verwenden. **`time`** (Shell builtin) zeigt falsche Realzeit an.
    - Usertime und Systemtime aller Threads eines Prozesses werden aufsummiert.
    -
```
timex auto
real        2.15
user        5.31
sys         0.32
```

Convex SPP - Besonderheiten

- ❏ Compiler
  - ❏ HP C Compiler
    - /bin/cc
    - Keine automatische Parallelisierung
    - Keine Optimierung, kein ANSI C
    - Bessere Codeerzeugung (instruction scheduling)
  - ❏ Convex C Compiler
    - **/usr/convex/bin/cc**
    - Für automatische Parallelisierung notwendig
    - Bessere Codeerzeugung (higher level optimizations)
- ❏ Linker
  - ❏ Parallele Programme müssen mit Convex Linker gebunden werden
    - **/usr/convex/bin/ld**

Convex SPP C Compiler

- ❏ /usr/convex/bin/cc
- ❏ Dokumentation
  - ❏ Manpage
  - ❏ Convex C Guide
  - ❏ Exemplar Programming Guide
- ❏ Kompatibel mit HP-C
  - ❏ HP und SPP Objektmodule können zusammengelinkt werden
  - ❏ Ausnahme: HP Objektmodule, die Graphikinstruktionen enthalten (werden für Networkcache Implementierung verwendet).
- ❏ Verschiedene Optimierungs-Level
- ❏ Parallelisierung

Convex SPP C Compiler

- ❏ ANSI-C kompatibel
- ❏ Spezielle Datentypen
  - ❏ 64 Bit Integer: **`long long`**
  - ❏ Spezielle Datentypen für die SPP Speicherklassen
  - ❏ Spezielle Datentypen für Synchronisierung
- ❏ Debugging und Profiling (mit voller Optimierung)
  - ❏ cxdb (compilieren mit -cxdb)
  - ❏ cxpa (compilieren mit -pa, -pab bzw. -par)

Convex SPP C Compiler

- ❏ Optimierung
  - ❏ Komandozeilenoptionen für Optimierungslevel
    - -no (default) Machine instruction level scalar optimizations
    - -O0 Basic block level scalar optimizations
    - -O1 Program unit level scalar optimizations, global register allocation
    - -O2 Global instruction scheduling, software pipelining, data location optimizations
    - -O3 Parallel optimizations
  - ❏ Optimierungslevel kann zusätzlich für einzelne Programmabschnitte gesetzt werden.
    - **`#pragma _CNX opt_level(-no|-O0|-O1|-O2|-O3)`**

Convex SPP C / Optimierung

- no / Machine instruction level scalar optimizations (default)
  - Optimierung nur innerhalb eines C Statements, um die arithmetischen Einheiten des Prozessors besser Auszunutzen (pipelining)
  - Ausnutzung von delay slots
  - Lokale Optimierung der Registerausnutzung
  - ...
- O0 / Basic block level scalar optimizations
  - Wie bei -no, aber innerhalb von Programmblöcken
  - Eliminierung redundanter Zuweisungen
  - Eliminierung gemeinsamer Teilausdrücke
  - Algebraische Vereinfachungen
  - ...

Convex SPP C / Optimierung

- O1 / Program unit level scalar optimizations, global register allocation
  - O0-Optimierung über Blockgrenzen
  - Herausziehen invarianter Ausdrücke aus Schleifen
  - ...
- O2 / Global instruction scheduling, software pipelining, data location optim.
  - Datenlokalität / Cache-Ausnutzung (Cachegröße muß bekannt sein)
  - Optimierung von Schleifen
  - ...
- O3 / Parallel optimization
  - ...nächster Abschnitt

Automatische Parallelisierung

- ❏ Leistungsfähigkeit der automatischen Parallelisierung
  - ❏ Eine automatische Parallelisierung kann nur geeignete Teile des Programmtextes erfassen. Sie ist geeignet, um Programme, ohne großen Programmieraufwand, teilweise parallel ablaufen zu lassen.
  - ❏ Eine automatische Parallelisierung ist i.a. **nicht** geeignet um die Leistungsfähigkeit einer parallelen Architektur voll oder weitgehend auszunutzen.
  - ❏ Durch spezielle Compilerdirektiven bzw. Pragma's kann eine weitgehende Parallelisierung erreicht werden. Der Begriff „automatisch" ist dann jedoch kaum noch gerechtfertigt.

Automatische Parallelisierung

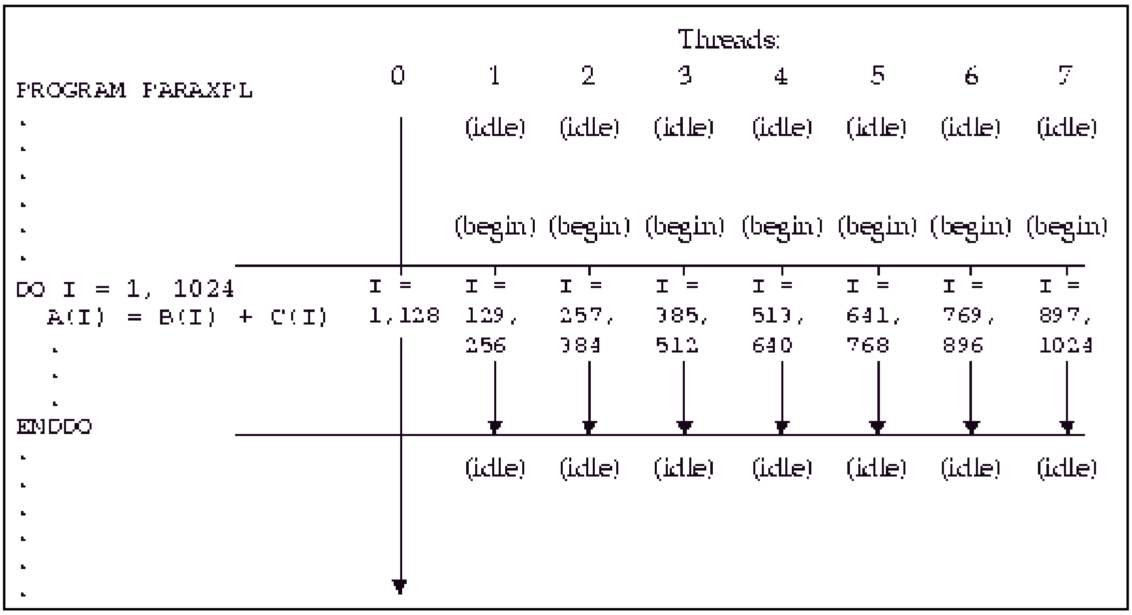
- ❏ Was wird parallelisiert?
  - ❏ Schleifen mit bekannter Iterationszahl,
    - falls der Compiler erkennt, daß eine Parallelisierung ohne Beeinflussung der Ergebnisse möglich ist (**starke Einschränkung**), oder
    - falls dem Compiler entsprechende Hinweise gegeben werden (Direktiven bzw. Pragma's).
    - Parallele Ausführung ganzer Programmabschnitte (nur mit Unterstützung durch die/den Programmierer/in).
- ❏ Wie wird parallelisiert?
  - ❏ Der Compiler verteilt die Iterationen einer Schleife gleichmäßig auf mehrere Threads eines Prozesses (max. ein Thread pro CPU - kein „oversubscribing").

Automatische Parallelisierung

- ❏ Ablauf eines parallelen Programms
  - ❏ Alle Threads werden zu Beginn der Programmausführung gestartet.
  - ❏ Alle Threads außer Thread 0 warten eine bestimme Zeit an einem Spinlock und blockieren sich dann.
  - ❏ Thread 0 führt den sequentiellen Code aus.
  - ❏ Zu Beginn eines parallelen Abschnitts sendet Thread 0 Signale an die wartenden Threads.
  - ❏ Alle Threads (auch Thread 0) bearbeiten ihren Anteil des parallelen Codes.
  - ❏ Alle Threads außer Thread 0 warten auf den nächsten parallelen Abschnitt.
  - ❏ Thread 0 führt den sequentiellen Code fort.

Automatische Parallelisierung

- ❏ Darstellung der Parallelisierung einer FORTRAN Schleife



Automatische Parallelisierung

- ❏ Beispiel:

  Die Schleife

```
for (i=0; i<n; i++) {
        a[i] = b[i] + c[i];
}
```

  wird umgewandelt in

```
for each available processor {
        for (i=tid*(n/nump); i<(n/nump)+tid*(n/nump), i++) {
                a[i] = b[i] + c[i];
        }
}
```

  wobei die Thread Id’s (tid) von 0 bis „Anzahl der verfügbaren Prozessoren (nump) - 1” eindeutig vergeben sind. Die innere Schleife wird von allen verfügbaren Prozessoren parallel ausgeführt.

Automatische Parallelisierung

- ❏ Schleifen können nicht automatisch parallelisiert werden, wenn einer der folgenden Fälle vorliegt:
  - ❏ Schleifen mit mehreren Ein- oder Ausgängen
  - ❏ Funktions- oder Prozeduraufrufe
  - ❏ I/O-Anweisungen
  - ❏ Iterationsanzahl unbekannt
  - ❏ Schleifenabhängigkeiten (Loop-carried dependencies (LCD))
  - ❏ Aliasing von Variablen oder Arrays
- ❏ Schleifen werden nicht automatisch parallelisiert, wenn
  - ❏ eine Parallelisierung sich nicht lohnt
  - ❏ dies explizit unerwünscht ist (#pragma _CNX no_parallel)

Loop-Carried Dependencies

❏ Backward LCD

```
for (i=1, i<n, i++) {
        a[i] = a[i-1] + b[i]
}
```

❏ Forward LCD

```
for (i=0, i<n-1, i++) {
        a[i] = a[i+1] + b[i]
}
```

❏ Apparent LCD

```
for ((i=0, i<n, i++) {
        a[i] = a[j[i]] + 1;
}
```


Loop-Carried Dependencies

❏ Output LCD

```
for (i=0, i<n, i++) {
        a[j[i]] = b[i];
}
```

Wenn sicher ist, daß die Schleife trotz dieser LDC parallelisiert werden kann (alle j[i] haben verschiedene Werte), kann dies dem Compiler mitgeteilt werden.

```
#pragma _CNX no_loop_dependence(a)

for ((i=0, i<n, i++) {
        a[j[i]] = b[i];
}
```

Aliasing von Arrays (Zeigern)

❑ Folgende Schleife kann nicht automatisch parallelisiert werden:

```
foo(a, b, c)
        float a[], b[];
        int c;
{
        int i;
        for (i=0; i<c; i++) {
                a[i] = b[i] * b[i];
        }
}
```

Es könnten der Funktion sich überschneidende Bereiche übergeben werden (eventuell in anderen Objektmodulen). Abhilfe:

- entweder mit **-alias array_args** übersetzen oder
- **pragma _CNX loop_parallel(ivar = i)** vor der Schleife einfügen.
- Bessere Lösung: Application Compiler

Automatische Parallelisierung

❑ Privatisierung von Variablen

Die Schleife

```
float, a[n], b[n], x[n], y[n], s
for (i=0, i<n, i++) {
        s = a[i] * b[i];
        x[i] = s * y[i];
}
```

kann nicht ohne weiteres parallelisiert werden, da sich die Threads die Variable **s** gegenseitig überschreiben können. Der Compiler legt daher eine private Kopie von **s** für jeden Thread an. Wird die Variable nach dem Schleifendurchlauf weiter verwendet, kopiert der Compiler den Wert des Threads, der die letzte Iteration bearbeitet hat, nach dem Schleifendurchlauf nach **s**. Der Compiler kann private Kopien von ganzen Datenstrukturen anlegen.

Unterstützte Parallelisierung (Schleifen)

- #pragma _CNX loop_private[(varlist)]

  Der Compiler nimmt keine automatische Variablenprivatisierung vor. Nur die angegebenen Variablen werden privatisiert. Im folgenden Beispiel kann der Compiler nicht entscheiden, ob **s** immer gesetzt wird, oder den Wert der vorhergehenden Iteration beibehält. **s** kann als loop_private deklariert werden, wenn sicher ist, daß immer eine Zuweisung erfolgt. Ist dies nicht so, kann die parallele Ausführung zu falschen Ergebnissen führen. Eine nicht unterstützte Parallelisierung ist daher nicht möglich.

```
#pragma _CNX loop_private(s)
for (i=0; i<n; i++) {
        if (a[i]>0) s = a[i];
        if (b[i]>0) s = b[i];
        b[i] = s * (a[i] + c[i]);
}
```


Unterstützte Parallelisierung (Schleifen)

- #pragma _CNX save_last
  Nur in Zusammenhang mit loop_private. Alle privatisierten Daten werden nach Ausführung der Schleife mit den Werten des Threads belegt, der die letzte Iteration berechnet hat. Dies ist notwendig, wenn privatisierte Daten nach Durchlauf der Schleife weiter verwendet werden. Das Pragma muß unmittelbar vor oder nach dem zugehörigen loop_private oder in der gleichen Zeile stehen. Beispiel:

```
#pragma _CNX loop_private(s), save_last
```

- #pragma _CNX prefer_parallel[(attribute-list)]

  prefer_parallel erzwingt keine Parallelisierung. Der Compiler parallelisiert nur, wenn eine Parallelisierung sicher ist. Eine Schleife wird jedoch auch dann parallelisiert, wenn sich eine Parallelisierung aus Sicht des Compilers nicht lohnt. Über die Attributliste kann die Parallelisierung der Schleife gesteuert werden.

Unterstützte Parallelisierung (Schleifen)

- #pragma _CNX loop_parallel(attribute-list)

  Die Parallelisierung der Schleife wird erzwungen. Der Compiler überprüft weder die Schleife auf Abhängigkeiten noch nimmt er eine Privatisierung von Variablen vor (Aufgabe der/des Programmierer(s)/in: über loop_private oder Speicherklassen). Das Attribut „ivar = var” muß angegeben werden.

- Attribute für prefer_parallel und loop_parallel

❏ ivar = var	Angabe der Iterationsvariablen (nur loop_parallel)
❏ threads	Parallelität über Threads (default)
❏ nodes	Parallelität über Nodes (1 Thread pro Node) Schachtelung mögl. (innere Schleife über Threads)
❏ chunk_size = n	Iterationszahl der vergebenen Abschnitte
❏ max_threads = n	max. Anzahl der verwendeten Threads
❏ ordered	alternatives Verteilungsschema (für ordered_section’s)

Unterstützte Parallelisierung (Task’s)

- Der Convex C Compiler unterstützt ebenfalls die parallele Ausführung von Programmabschnitten. Beispiel:

```
#pragma _CNX task_private(i, help)
#pragma _CNX begin_tasks
        for (i=0; i<n; i++)
                help[i] = a[i];
        for (i=0; i<n; i++)
                a[func1(help[i])] = help[i]
#pragma _CNX next_task
        for (i=0; i<m; i++)
                help[i] = b[i];
        for (i=0; i<m; i++)
                b[func2(help[i])] = help[i]
#pragma _CNX end_tasks
```

Unterstützte Parallelisierung (Task's)

- Der Compiler wandelt den Code in eine Schleife mit (im Fall des Beispiels) Iterationszahl 2 um. In jedem Schleifendurchlauf wird genau einer der Abschnitte ausgeführt. Diese Schleife wird parallelisiert. Dabei werden
  - keine Variablenabhängigkeiten überprüft,
  - keine Variablen privatisiert und
  - keine Synchronisationen vorgenommen.
- begin_tasks unterstützt folgende Parameter. Die Bedeutung entspricht der bei der Schleifenparallelisierung:
  - nodes
  - threads
  - max_threads
  - ordered

Unterstützte Parallelisierung

- Gegenseitiger Ausschluß (critical_section)
  - #pragma _CNX critical_section[(gate)]
    #pragma _CNX end_critical_section
- Geordnete Abschnitte (ordered_section)
  - #pragma _CNX ordered_section[(gate)]
    #pragma _CNX end_ordered_section
  - nur in Verbindung mit loop_parallel(ordered)
- Bemerkungen zu critical_section und ordered_section
  - Zusätzlich kann eine Gate-Variable (gate_t) angegeben werden, um zwischen verschiedenen kritischen bzw. geordneten Abschnitten zu unterscheiden. Sie muß von der/dem Programmierer/in initialisiert werden.

```
static gate_t gate;
alloc_gate(&gate);
```

Automatische Parallelisierung

- ❏ Aufruf von Compiler und Linker
  - ❏ Convex Compiler:
    -O3
  - ❏ weiter Compiler Flags
    - -or none   unterdrückt Optimierungsreport
  - ❏ Convex Linker (etwas aufwendig):
    -L/usr/convex/all -L/usr/convex/all/spp1 +tm spp1 +parallell
    /usr/convex/all/crt0.o -lcps -lc
  - ❏ daher besser über cc Frontend:
    /usr/convex/bin/cc -o test test.o
  - ❏ weitere Linker Flags:
    - +min n   minimale Anzahl von Threads
    - +max n   maximale Anzahl von Threads (<= Anzahl der Proz.)
    - +over   Oversubscription (wirkungslos)

Automatische Parallelisierung

- ❏ Beispiel für ein makefile:

```
CC=/usr/convex/bin/cc
LD=/usr/convex/bin/cc

auto:   auto.o
        ${LD} -o $@ $@.o

auto.o: auto.c
        ${CC} -O3 -c $<
```

Automatische Parallelisierung

- ❑ Beispiel: Matrixmultiplikation (auto.c)

```
for (i=0; i<1000; i++) {
  for (j=0; j<1000; j++) {
    for (k=0; k<1000; k++) {
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
  }
}
```

- ❑ Die Schleifenoptimierung ergibt den folgenden Code.

```
for (jout=0; jout<1000; jout+=jblock) {
  for (kout=0; kout<1000; kout+=kblock) {
    for (i=0; i<1000; i++) {
      for (j=jout; j<jout+jblock; j++) {
        for (k=kout; k<kout+kblock; k++ {
          c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
      }
    }
  }
}
```

- ❑ Die äußerste Schleife wird parallelisiert.

Automatische Parallelisierung

- ❑ Zeitmessung für das Beispiel: Matrixmultiplikation in Sekunden (Werte vom letzten Jahr in Klammern)

❑ -no	895.3	(2271.6)	
❑ -O0	670.3	(1425.3)	
❑ -O1	401.4	(535.2)	
❑ -O2	51.6	(308.9)	
❑ -O3	12.7	(55.6)	auf 8 Prozessoren

- ❑ **Übungsaufgabe**
  - ❑ In der Datei /home/gsm/i441/Beispiele/auto/auto.bsp liegen einige Beispiele für Schleifen.
    - Welche davon können automatisch parallelisiert werden?
    - Welche nicht und warum?
    - **Bitte nicht ausprobieren!**

Application Compiler

- ❏ Application Compiler unterstützt C und Fortran
- ❏ Interprozeduale Optimierung
  - ❏ Über Filegrenzen (apc analysiert den gesamten Source-Baum)
  - ❏ Inlining von Funktionen
  - ❏ Cloning von Funktionen
    - • Duplizierung und unterschiedliche Optimierung für verschiedene Parameter
  - ❏ Parallele Loops mit Funktionsaufrufen
  - ❏ Weitergehende automatische Schleifenparallelisierung möglich
  - ❏ ...
- ❏ Seit zwei Wochen verfügbar, noch keine Erfahrungen!

Convex SPP Speicherklassen

- ❏ thread_private
  - ❏ Private Daten eines Threads.
  - ❏ Physikalische Speicherklasse: Node private
  - ❏ Realisiert durch Compiler (kein Zugriffsschutz unter den Threads eines Prozesses).
  - ❏ Statisch: Jeder Thread greift auf seine private Kopie unter der gleichen virtuellen Adresse zu.
  - ❏ Dynamisch: Nur für den anfordernden Thread „zugreifbar”. Nur sinnvoll in parallelen Abschnitten.

Convex SPP Speicherklassen

- node_private
  - Zugreifbar für die Threads eines Prozesses auf einem Node.
  - Physikalische Speicherklasse: Node private
  - Statisch: Alle Threads greifen unter der gleichen virtuellen Adresse auf die Kopie ihres Nodes zu.
  - Dynamisch: Für alle Threads des Nodes zugreifbar, dem der anfordernde Thread zugeordnet ist.
- near_shared
  - Zugreifbar für alle Threads eines Prozesses.
  - Physikalische Speicherklasse: Subcomplex global
  - Statisch: Liegt auf Node 0 (?) / Wenig sinnvoll!
  - Dynamisch: Liegt auf dem Node, dem der anfordernde Thread zugeordnet ist.

Convex SPP Speicherklassen

- far_shared
  - Zugreifbar für alle Threads eines Prozesses.
  - Physikalische Speicherklasse: Subcomplex global
  - Statisch und dynamisch:
    Wird seitenweise (pagesize: 4 kilobyte) auf alle Nodes des Subcomplexes verteilt (nach der „round robin” Strategie).
- block_shared
  - Zugreifbar für alle Threads eines Prozesses.
  - Physikalische Speicherklasse: Subcomplex global
  - Statisch: Statische Vereinbarung nicht möglich.
  - Dynamisch: Wird auf allen Nodes, auf denen Threads des Prozesses laufen, gleichmäßig in zusammenhängenden Bereichen verteilt.

Convex SPP Speicherklassen

- Verwendung von near, far und block_shared
  - near_shared
    - Daten, die von allen Threads aus erreichbar sein müssen, aber von dem anfordernden Thread häufiger zugegriffen werden. Die Zugriffs-zeiten sind für den anfordernden Thread kürzer, da die Daten auf dem gleichen Node liegen.
  - far_shared
    - Geeignet für statistisch gleichverteilte Zugriffe von allen Nodes aus. Die Daten sind gleichmäßig auf allen Nodes verteilt.
  - block_shared
    - Geeignet für Daten, die von allen Threads aus zugreifbar sein müs-sen, aber blockweise von bestimmten Threads stärker benötigt wer-den.

Convex SPP Speicherklassen

- Einbindung in C

```
#include <spp_prog_model.h> /* in /usr/include */
```

  - Dort sind definiert:

```
thread_private
node_private
near_shared
far_shared

THREAD_PRIVATE_MEM
NODE_PRIVATE_MEM
NEAR_SHARED_MEM
FAR_SHARED_MEM
BLOCK_SHARED_MEM

void *memory_class_malloc(size_t bytes, int class);
```

Convex SPP Speicherklassen

- Beispiele:

```
thread_private int = 10;
node_private double;
far_shared float[1024];

thread_private double* ptr;
ptr = (double *)memory_class_malloc(80000, BLOCK_SAHRED_MEM);

/* 2 Nodes: 80000/4096/2 = 9.8... -> 10 Pages pro Node
 *          angefordert werden 10*4096*2 = 81380 Bytes
 */
```

- Einsatz der Speicherklassen
  - Manuelle Parallelisierung
  - Unterstützte automatische Parallelisierung
  - Programme mit > 4 GB Speicherbedarf

Parallele Programmierung (CPS)

- Compiler Support Library (CPS)
  - Entworfen zur Unterstützung parallelisierender Compiler
  - Explizite Programmierung ist auch möglich
  - Weniger komfortabel als z.B. PThreads
- Compilieren und Linken
  - Analog zur automatischen Parallelisierung
    (siehe Makefile im Abschnitt „Automatische Parallelisierung”)
  - +parallel [+min n] [+max n]
  - +over wirkungslos
  - Automatische Parallelisierung weiterhin möglich (im Rahmen der verbleibenden Prozessorkapazitäten)

Parallele Programmierung (CPS)

- ❏ Funktionsprinzip
  - ❏ Analog zur automatischen Parallelisierung
  - ❏ Auf jedem verfügbaren Prozessor wird zu Programmstart ein Thread generiert (event. eingeschränkt durch +min oder +max) und bei Bedarf aktiviert.
  - ❏ Es ist kein Oversubscribing der Prozessoren mit Threads möglich (alle diesbezüglichen Parameter werden z.Z. ignoriert).
  - ❏ Die Threads der CPS werden 1:1 auf Kernelthreads „abgebildet".
  - ❏ Jeder Thread wird durch 2 Identifikatoren eindeutig gekennzeichnet:
    - Kernel Thread ID (eindeutig innerhalb eines Prozesses)
    - Spawn Thread ID (eindeutig innerhalb eines Spawn-Konstruktes (0 - nsthreads -1))

Parallele Programmierung (CPS)

- ❏ Funktionen zur Compilerunterstützung

```
#include <cps.h>

typedef struct {
       int node;              /* node to place threads on */
       int min:               /* min number of threads */
       int max;               /* max number of threads */
       int threadscope;       /* div. Flags */
} spawn_sym_t;

int __cps_spawn(spawn_sym_t *arg);
int __cps_join(void);
```

  - ❏ Werte für Node:

```
CPS_ANY_NODE
CPS_SAME_NODE
CPS_DIFFERENT_NODE
```

Parallele Programmierung (CPS)

- ❑ Werte für Threadscope:
  CPS_NODE_PARALLEL
  CPS_THREAD_PARALLEL
  CPS_OVER_SUBSCRIBE
  CPS_IGNORE_STACKSCOPE

- ❑ Variablen zur Compilerunterstützung

```
__cps_stid              /* spawn thread id */
__cps_ktid              /* kernel thread id */
__cps_nsthreads         /* number of spwan threads */
__cps_node_id           /* node number */
__cps_node_cpus
__cps_node_nthreads
__cps_is_parallel
__cps_plevel
...
```


Parallele Programmierung (CPS)

- ❑ Parallelisierung durch den Compiler.
  Aus:

```
int i,j;

for (i=0; i<512; i++)
  for(j=0; j<512; j++)
    a[i][j] += b[i][j];
```

Wird:

```
#include <cps.h>
spawn_sym_t sp1 = {
  CPS_ANY_NODE,
  1,
  16,
  CPS_THREAD_PARALLEL
};
thread_private int i,j;
thread_private int wrk, low, high;
int n;

n = __cps_spawn(&sp1);
wrk = (512+n-1)/n;
low = wrk * __cps_stid;
high = (low+wrk) > 512 ? 512 : (low+wrk);
for (i=low; i<high; i++)
  for(j=0; j<512; j++)
    a[i][j] += b[i][j];
__cps_join();
```

Parallele Programmierung (CPS)

❑ cps_ppcall: Version für die manuelle Programmierung

```
int cps_ppcall(spawn_sym_t *arg,
               void (*func)(void *), void *ptr);
```

  ❑ Analog zu __cps_spawn.

  ❑ Alle Threads (auch der Aufrufende) arbeiten die Funktion **func** mit Parameter **ptr** ab.

  ❑ Nach Abarbeitung der Funktion (return) rufen sie implizit __cps_join auf.

❑ Zugriffsfunktionen für die CPS Variablen

```
int cps_stid(void);
int cps_ktid(void);
int cps_nsthreads(void);
...
```


Parallele Programmierung (CPS)

❑ Symmetrische Threads

  ❑ __cps_spawn und cps_ppcall erzeugen symmetrische Threads. Alle bearbeiten den gleichen Code und synchronisieren sich danach implizit.

❑ Asymmetrische Threads

  ❑ Threads werden einzeln mit individuellen Aufgaben erzeugt und terminieren unabhängig voneinander.

```
int cps_thread_create(const int *node,
                      void (*func)(void *), void *ptr);
int cps_thread_exit(void);
int cps_thread_wait(const int *flag);
```

Parallele Programmierung (CPS)

❏ Beispiel

```
int node = CPS_ANY_NODE;
int on = 1;

void thread(arg)
     void *arg;
{
  /* ... */
  cps_thread_exit();
}

for (i=0; i<MAX; i++) {
  ktid = cps_thread_create(&node, thread,
                           (void *)0);
  if (ktid == -1)
    perror("thread create");
  else
   printf("I'v created thread with ktid %d\n",
          ktid);
}

cps_thread_wait(&on);
```


Parallele Programmierung (CPS)

❏ cps_thread_create
- node: Node Id, CPS_SAME_NODE oder CPS_ANY_NODE
- Es wird keine Spawn-Thread-Id (stid) gesetzt (__cps_stid hat den Wert -1).
- Rückgabewert ist die Kernel-Thread-Id.
- cps_thread_exit wird implizit nach Funktionsreturn aufgerufen.

❏ cps_thread_exit
- Terminiert den aufrufenden asymmetrischen Thread.

❏ cps_thread_wait
- flag =1: wartet bis alle asymmetrischen Threads des Prozesses terminiert sind.
- flag = 0: gibt die Anzahl der aktiven asymmetrischen Threads des Prozesses zurück.
- Nicht mit Flag=1 von einem asymmetrischen Thread aus aufrufen.

Parallele Programmierung (CPS)

❏ Zusammenfassung:

  ❏ Geeignet für automatische Parallelisierung.

  ❏ Wenig für manuelle Programmierung, wenig komfortabel, geringe Funktionalität.

   • Kein Oversubscribing
   • Kein wait auf explizite Threads
   • Umständliches Funktionsinterface
   • Kein Suspendieren von Threads (nicht in CPS!)
     (kann cnx_thread_block / cnx_thread_unblock verwendet werden?)
   • ...

Parallele Programmierung (CPS)

❏ CPS Synchronisationsmechanismen

  ❏ Low level

   • Cache basierte Semaphoren (cache_sema_t)
     Geringe Latenz (auf einem Node)
   • Speicher basierte Semaphoren (mem_sema_t)
     Geeignet für Multinode-Anwendungen
   • Atomare Zähler und „fetch_and_clear” <u>oder</u>
     lock (binäre Semaphore)
   • Vorsicht bei Rückgabewert „-1”
   • Beispiele im Abschnitt über Kernel Thread Programmierung

  ❏ High level

   • Mutex (cps_mutex_t)
   • Barrier (barrier_t)

CPS Semaphore

```
c_init32(cs, val)                m_init32(ms, val)
c_free32(cs)                     m_free32(ms)

c_fetch32(cs)                    m_fetch32(ms)
c_fetch_and_inc32(cs)            m_fetch_and_inc32(ms)
c_fetch_and_dec32(cs)            m_fetch_and_dec32(ms)
c_fetch_and_clear32(cs)          m_fetch_and_clear32(ms)
c_fetch_and_set32(cs, newval)
c_fetch_and_add32(cs, addval)

c_lock(cs)                       m_lock(ms)
c_unlock(cs)                     m_unlock(ms)
c_cond_lock(cs)                  m_cond_lock(ms)
```

```
mem_sema_t *ms;
cache_sema_t *cs;
int *val, newval, addval;
```


CPS Mutex und Barrier

```
int cps_mutex_alloc(cps_mutex_t *mutex);
int cps_mutex_lock(cps_mutex_t *mutex);
int cps_mutex_trylock(cps_mutex_t *mutex);
int cps_mutex_unlock(cps_mutex_t *mutex);
int cps_mutex_free(cps_mutex_t *mutex);

int cps_barrier_alloc(barrier_t *barrier);
int cps_barrier(barrier_t *barrier, const int *n);
int cps_barrier_free((barrier_t *barrier);
```

- Bemerkungen
  - Es sind nicht alle Mechanismen auch außerhalb des CPS Programmiermodells einsetzbar. Bei direkter Programmierung mit cnx_thread_create können nur die Low-level Semaphoren eingesetzt werden.
  - Die Mutex Funktionen sind derzeit so implemetiert, daß zunächst ein Spinlock (active wait) erfolgt und sich der Thread nach einigen Versuchen suspendiert. (siehe `cps_wait_attr()`)

CPS Mutex Beispiel

```
/* Initialisierung im sequentiellen Programmteil */

cps_mutex_t mutex;

cps_mutex_alloc(&mutex);

/* und im parallelen Teil - z.B. nach einem thread_create */

cps_mutex_lock(&mutex);

/* kritischer Abschnitt */

cps_mutex_unlock(&mutex);
```


CPS Barrier Beispiel

```
/* Initialisierung im sequentiellen Programmteil */

barrier_t barrier;
int barr_count;

cps_barrier_alloc(&barrier);
barr_count = 2;

/* und im parallelen Teil - z.B. nach einem thread_create     */
/* es laufen 2 unsynchronisierte Thread, die einen bestimmten */
/* Punkt erreichen müssen, ehe der Programmablauf fortgesetzt */
/* werden kann                                               */

cps_barrier_wait(&barrier, &barr_count);

/* nachdem beide threads cps_barrier_wait aufgerufen haben */
/* werden sie (synchronisiert) fortgesetzt                 */
```

Parallele Programmierung (Systemcalls)

- ❏ Systemcalls

```
#include <sys/cnx_thread.h>
#include <sys/cnx_pattr.h>
#include <spp_prog_model.h>
```

  - ❏ cnx_thread_create

```
tid_t cnx_thread_create(node_t node, void *sp,
                        void (*func)(int),
                        unsigned int flags);
```

    - Der neue Thread führt die Funktion **`func`** aus. Als einziger Parameter wird die Kernel-Thread-Id übergeben.
    - Der Thread muß vor dem Rücksprung aus **`func`** cnx_thread_exit aufrufen.
    - Ein Stack muß vor dem Erzeugen eines Threads angefordert werden.
    - Als Flag CNX_USER_THREAD angeben.

Parallele Programmierung (Systemcalls)

- ❏ cnx_thread_exit

```
int cnx_thread_exit();
```

  - Beendet den aufrufenden Thread.
  - Falls es sich um den letzten Thread eines Prozesses handelt, wird der Prozeß terminiert.

- ❏ cnx_thread_self

```
tid_t cnx_thread_self();
```

  - Gibt die Kernel-Thread -ID des aufrufenden Prozesses zurück.

- ❏ cnx_thread_block

```
int cnx_thread_block(unsigned int timeout_val,
                     unsigned int *result
```

  - Blockiert den aufrufenden Thread.
  - Der Prozeß wird nach einem Timeout von **`timeout`** (>0) Microsekunden oder durch cnx_thread_unblock deblockiert.

Parallele Programmierung (Systemcalls)

- Ist **`result`** != NULL, wird dort ein Wert zurückgegeben, der bei cnx_thread_unblock übergeben wird.
- Wurde cnx_thread_unblock vor cnx_thread_block aufgerufen, terminiert cnx_thread_block sofort.

❑ cnx_thread_unblock

```
int cnx_thread_unblock(tid_t thread_id, unsigned int value,
                        unsigned int *queue_length);
```

- Deblockiert den Thread mit der Kernel-Thread-Id **`thread_id`.**
- Der Wert **`value`** wird dem deblockierten Thread über cnx_thread_block übergeben bzw. an eine Warteschlange von cnx_thread_unblock-Aufrufen für diesen Prozeß angehängt.
- Ist queue_length != NULL, wird die Länge der Warteschlange zurückgegeben.

Parallele Programmierung (Systemcalls)

❑ Was fehlt?
- cnx_thread_wait
- cnx_thread_suspend
- cnx_thread_continue

❑ Systemcalls zur Konfiguration des Systems

❑ cnx_setpattr, cnx_getpattr, cnx_settattr, cnx_gettattr

❑ Siehe Beispiel

❑ Aufruf von Compiler und Linker

❑ Convex Compiler:
max. mit -O2 optimieren

Parallele Programmierung (Systemcalls)

- ❏ Convex Linker (ebenfalls einfacher über cc):
  -L/usr/convex/all -L/usr/convex/all/spp1 +tm spp1 +parallel +max 1
  /usr/convex/lib/crt0.o -lcps -lc
- ❏ weitere Linker Flags:
  - +over Oversubscription (erlaubt mehr Kernel-Threads als Prozessoren)

Parallele Programmierung (Systemcalls)

- ❏ Beispiel für ein makefile:

```
CC=/usr/convex/bin/cc
LD=/usr/convex/bin/cc
LD_OPTS=-Wl,+parallel,+max,1
LD_LIBS=-lcps -lc

auto:   sema.o
        ${LD} ${LD_OPTS} -o $@ $@.o ${LD_LIBS}

auto.o: sema.c
        ${CC} -O2 -c $<
```

Parallele Programmierung (Systemcalls)

- Synchronisationsmechanismen
  - Die Synchronisationsmechanismen der CPS-Library können teilweise verwendet werden (Low-level Semaphore: c_..., m_...).
  - Die gate, barrier, cps_mutex und cps_barrier Funktionen können nicht verwendet werden.
  - Die Synchronisationsfunktionen aus der libail können verwendet werden. Siehe “man ail”; low-level Funktionen; dürfen nicht *gecached* sein.
- Zusammenfassung
  - Im Systemcallinterface fehlen einige wichtige Funktionen, die im darunterliegenden MACH noch unterstützt werden.
  - Wenig komfortabel.

Beispiel für cnx_thread...

```
cache_sema_t sema1=NULL

thread(arg)
int arg;
{
  for (i=0; i<8; i++) {
    prim_sema_p(&sema1);
    printf("thread %d has sema\n", arg);
    prim_sema_v(&sema1);
  }
  cnx_thread_exit();
}

main()
{
  int val=0;

  set_nthreads(CNX_PATTR_THREAD_NO_LIMIT);
  prim_sema_init(&sema1, &val);

  for (i=0; i<8; i++)
    {
      sp[i] = (caddr_t) malloc (8*1024*1024);
      ktid = cnx_thread_create(0, sp[i], thread,
                               CNX_USER_THREAD);
    }

  prim_sema_v(&sema1);
  cnx_thread_exit();
}
```

Beispiel für cnx_thread...

```
set_nthreads(nthreads)
int nthreads;
{
  long pid;
  cnx_pattributes_t pattributes;

  pid = getpid();

  /*
  pattributes.pattr_maxth = nthreads;
  cnx_setpattr(pid,CNX_PATTR_MAXTH,&pattributes);
  */

  pattributes.pattr_oversubscription = 1;
  cnx_setpattr(pid, CNX_PATTR_OVERSUBSCRIPTION,
               &pattributes);
}
```


Beispiel für c_fetch_and...

```
int prim_sema_init(sema, val)
  cache_sema_t *sema;
  int *val;
{
  int ret;

  ret = c_init32(sema, val);
  if (ret == -1 && *sema == NULL)
    return -1;
  return 0;
}

int prim_sema_free(sema)
  cache_sema_t *sema;
{
  int ret;

  ret = c_free32(sema);
  if (ret == -1 && *sema == NULL)
    return -1;
  return 0;
}
```

P und V - Ein Versuch (active wait)

```
int prim_sema_p(sema)
  cache_sema_t *sema;
{
  int ok=0, ret;

  while (!ok) {
    ret = c_fetch_and_dec32(sema);
    if (ret == -1 && *sema == NULL)
      return -1;
    if (ret > 0)
      ok = 1;
    else {
      ret = c_fetch_and_inc32(sema);
      if (ret == -1 && *sema == NULL)
        return -1;
    }
  }
  return 0;
}

int prim_sema_v(sema)
  cache_sema_t *sema;
{
  int ret;

  ret = c_fetch_and_inc32(sema);
  if (ret == -1 && *sema == NULL)
    return -1;
  return 0;
}
```


P und V - Ein Versuch (active wait)

- Was funktioniert an dieser Lösung nicht?
  - Ein „Live-Lock" ist möglich!
    - Die unteilbare (atomare) Zählvariable kann ständig kleiner Null sein.
  - Ein atomarer Zähler ist noch keine Semaphore!
    - Wofür können atomare Zähler dann eingesetzt werden?
- Wie kann das Problem gelöst werden?
  - Zusätzliches Locking, etwa mit einem Spinlock.
    - Warum dann noch einen atomaren Zähler?
      Je nach Implementierung (relativ problemlos bei Lösungen mit sich suspendierenden Threads), können Teile des Codes aus dem Lokking herausgezogen werden. Dann ist Atomarität wieder gefordert.

P und V - Ein Versuch (active wait)

- ❏ Realisierung des Locking
  - ❏ Über lock und unlock
    - Vorsicht: nicht auf der gleichen Semaphore (c_sema_t)!
  - ❏ Eigenes Locking über fetch_and_clear (analog zu test_and_set) und fetch_and_inc.
    - Warum ergeben sich nicht ähnliche Probleme wie bei prim_sema_p und prim_sema_v?
- ❏ **Übungsaufgabe**
  - ❏ Erweitern Sie die vorgestellten primitiven Semaphoren (/home/gsm/i441/Beispiele/threads/prim_sema.c) so,
    - daß sie livelock-frei sind (locking),
    - daß sie nicht aktiv warten, sondern sich blockieren (cnx_thread_block in sema_p, cnx_thread_unblock in sema_v) und

P und V - Ein Versuch (active wait)

- daß die kritischen Abschnitte innerhalb des Locking möglichst klein sind (nicht unbedingt erforderlich!).

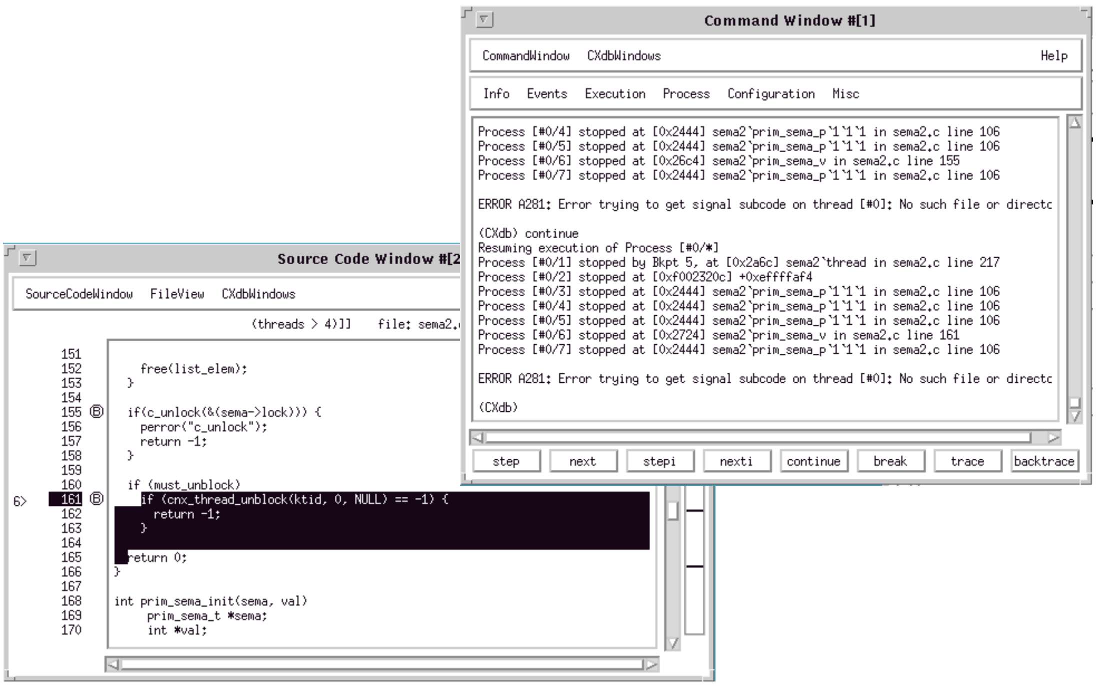
- ❏ Dabei ist zu beachten:
  - Es ist einfacher, immer nur den am längsten wartenden Thread zu deblockieren (first come first serv).
  - Wenn Ihre Lösung mit deutlich mehr Threads als Prozessoren (etwa doppelt so viele) läuft, ist dies ein Hinweis darauf, daß sie richtig sein könnte.
  - Sie benötigen einen eigenen Datentyp Semaphore. Eine Struktur mit in etwa folgenden Elementen:

```
struct my_sema {
    c_sema_t     counter;
    c_sema_t     lock;
    thread_list  *list;      /* verkettete Liste der blockierten
                                   threads */
};
```

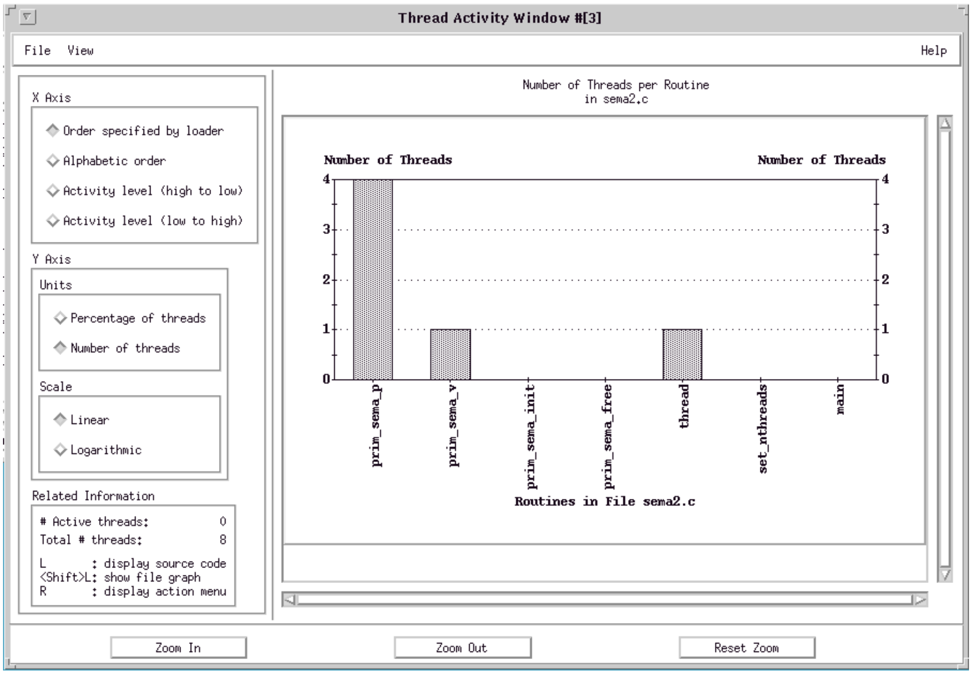
Cxdb - Debugger

- ❑ Sourcelevel Debugger
  - ❑ C und Fortran
  - ❑ Multithread Support
  - ❑ Debuggen von optimierten Code
  - ❑ Debuggen einzelner Threads
  - ❑ ...
- ❑ X Window Interface
  - ❑ Command-, Source-, Threadwindow...
- ❑ Compilieren mit -cxdb
  - ❑ Aufruf: **`/usr/convex/bin/cxdb program [args]`**

Cxdb - Debugger



Cxdb - Debugger



Cxpa - Profiler

- ❑ Profiler
  - ❑ C und Fortran
  - ❑ Multithread Support
  - ❑ Profiling von optimierten Code
  - ❑ ...
- ❑ X Window Interface
  - ❑ Command-, Source-, 2 und 3D Display, Profiling Reports
- ❑ Compilieren mit -cxpa
  - ❑ Aufruf: **`/usr/convex/bin/cxpa program [args]`**

Cxpa - Profiler