Parallelrechnerseminar IWR
TU Dresden

Parallelisierung
Programmieren im *shared memory*
Paralleles Objektorientiertes Programmieren

N. Schmeißer*
Forschungszentrum Rossendorf e.V.,
Abt. Kommunikation und Datenverarbeitung

21. November 1996

*Telefon: (+49 351) 260/2207, E-Mail: nils@schmeisser.com,
URL: http://www.fz-rossendorf.de/FVTK/MITARB/schmei

Inhaltsverzeichnis

1 Rechnerklassen **3**

2 Parallelisierung **3**
- 2.1 Sequentielle Optimierung 4
- 2.2 motivierendes Beispiel 5
- 2.3 Datenabhängigkeiten 6
- 2.4 Code-Parallelisierung 7
- 2.5 Schleifenparallelisierung 8
- 2.6 Kommunikation 9

3 Paralleles Rechnen im shared memory **10**
- 3.1 Beispiel: 1D-FFT auf SPP1000 11
- 3.2 Vermeidung von Zugriffskonflikten 12
- 3.3 dead-locks 13
  - 3.3.1 Decker-Algorithmus 13
  - 3.3.2 Algorithmus von Dijkstra 14
  - 3.3.3 Algorithmus von Knuth 15

4 Paralleles Objektorientiertes Programmieren **16**

1 Rechnerklassen

	ein Datum	viele Daten
eine Instruktion	**SISD**	**SIMD**
viele Instruktionen	**MISD**	**MIMD**

SISD von Neumann-Rechner, klassischer sequentieller Rechner
SIMD Vektorrechner → Convex C-Serie
MIMD "echter" Parallelrechner → SPP1000, WS-Cluster
MISD ?

2 Parallelisierung

Parallelisierung läuft in 2 Phasen:

- Sequentielle Optimierung:
  - Konsistenzprüfung
  - Kontroll- und Datenstrukturen analysieren
  - Codequalität
  - Laufzeitstatistik
  - Optimieren- und Restrukturieren

- Parallelisieren:
  - Datenaufteilung
  - Schleifenparallelisierung
  - Kommunikation
  - Testen
  - Optimieren- und Restrukturieren

2.1 Sequentielle Optimierung

Beispiel für Optimierung:

	cp:=cos(phi);
	sp:=sin(phi);
*hx:=x*cos(phi)-y*sin(phi);*	*hx:=x*cp-y*sp;*
*hy:=x*sin(phi)+y*cos(phi);*	*hy:=x*sp+y*cp;*
x:=hx;	*x:=hx;*
y:=hy;	*y:=hy;*

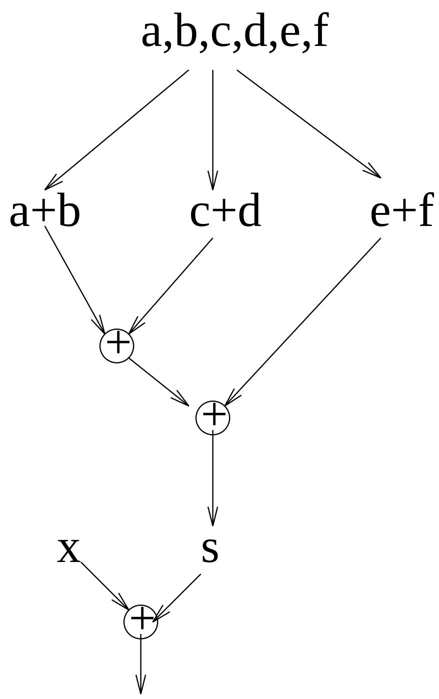
Beispiel für Restrukturierung:

if (i=0) then	*if (i=0) then*
do1;	*do1*
if (i=1) then	*else*
do2;	*do2;*

→ scalar replacement, loop fusing, loop unrolling

2.2 motivierendes Beispiel

Beispiel:
$s = a + b + c + d + e + f = ((a + b) + (c + d)) + (e + f)$
$t = 2s + x$



$\Rightarrow$ **Datenabhängigkeiten bestimmen**

2.3 Datenabhängigkeiten

Eingabemenge *IN(S)***:** Menge aller Variablen, deren Werte in S benötigt

Ausgabemenge *OUT(S)***:** Menge aller Variablen, deren Werte in S geändert werden können

Ausführungsreihenfolge: Sei S eingeschlossen in eine Schleife mit Laufvariable i, dann bedeutet $S_1^\nu \Theta S_2^\nu$, S_1 wird beim Durchlauf $i = \nu$ vor S_2 ausgeführt.

Beispiel:

S_1="c:=a+b;"	$IN(S_1)=\{a,b\}$, $OUT(S_1)=\{c\}$
S_2="d:=a+c;"	$IN(S_2)=\{a,c\}$, $OUT(S_2)=\{d\}$
	$S_1 \Theta S_2$

flow-dependent $S_1 \delta S_2$**:** $\exists x \in OUT(S_1) \wedge x \in IN(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k : (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
Bsp.: $S_1 \delta S_2$

anti-dependent $S_1 \bar{\delta} S_2$**:** $\exists x \in IN(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k : (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
Bsp.: $S_1 =$"$d := a + c$", $S_2 =$"$c := a + b$" $\rightarrow S_1 \bar{\delta} S_2$

output-dependent $S_1 \delta^0 S_2$**:** $\exists x \in OUT(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k : (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
Bsp.: $S_1 =$"$d := a + c$", $S_2 =$"$d := a + b$" $\rightarrow S_1 \delta^0 S_2$

data-dependent $S_1 \delta^* S_2$**:** Falls $S_1 \delta S_2 \vee S_1 \bar{\delta} S_2 \vee S_1 \delta^0 S_2$

Die Parallelisierung kann erfolgen, wenn zwei Anweisungen S_1 und S_2 datenunabhängig sind.

2.4 Code-Parallelisierung

Beispiel:

- Umstrukturierung des sequentiellen Codes:

s:=((a+b)+(c+d))+e+f); $\rightarrow$	*h1:=a+b;* $\{ S_1 \}$
	h2:=c+d; $\{ S_2 \}$
	h3:=e+f; $\{ S_3 \}$
	h4:=h1+h2; $\{ S_4 \}$
	s:=h4+h3; $\{ S_5 \}$
*t:=2*s+x;* $\rightarrow$	*t:=2*s+x;* $\{ S_6 \}$

- Parallelisierung:

P_0	P_1	P_2
h1:=a+b;	*h2:=c+d;*	*h3:=e+f;*
sync(P_0,P_1);	*sync*(P_0,P_1);	
h4:=h1+h2;		
sync(P_0,P_2);		*sync*(P_0,P_2);
s:=h4+h3;		
*t:=2*s+x;*		

- Probleme:
  - S_1, S_2 und S_3 werden zwar parallel ausgeführt, aber Synchronisation(=Kommunikation) notwendig
  $\rightarrow$ Beachten des *Laufzeitstatistik*
  - aus Berechnungsgraph und Laufzeitstatistik einen (sub)optimalen *Schedule* aufbauen

2.5 Schleifenparallelisierung

Beispiel:

```
var a,b,c,d:array[0..MAXINDEX] of real;

for i:=0 to n do begin
  c[i]:=a[i]+b[i];      { S1 }
  d[i]:=a[i]*b[i];      { S2, S2 datenunabhängig von S1 }
end;
```

⇒ parallelisierbar

P_1	P_2
for i:=0 to n do begin *c[i]:=a[i]+b[i]; { S1 }* *end;*	*for i:=0 to n do begin* *d[i]:=a[i]*b[i]; { S2 }* *end;*

c[0]:=...;
for i:=1 to n do begin
  c[i]:=a[i]+b[i];   { *S1* }
  *d[i]:=c[i-1]*a[i];*   { *S2*, $S_1^{i-1} \delta S_2^i$ }
end;

⇒ **nicht** parallelisierbar (aber pipelining möglich)

2.6 Kommunikation

- Shared Memory: zwei Prozesse teilen sich einen gemeinsamen Speicher

P0	*P1*	
var a:integer;	*var a:integer;*	*die beiden Deklarationen*
	var b:integer;	*beziehen sich auf eine einzige*
		Instanz der Variablen a im
		Speicher }
...	...	
a:=1;		
sync(P0,P1);	*sync(P0,P1);*	← *Synchronisation nötig*
	b:=a+1;	

- Message Passing: zwischen Prozessen werden Nachrichten verschickt

P0	*P1*	
private;	*private;*	
var a:integer;	*var a,b:integer;*	*für jeden Prozeß existiert eine*
		lokale, nach außen nicht sichtbare
		Kopie der Variablen a
a:=1;	...	
send(a,P1);	*recv(a,P0);*	*"blocking" operation, d.h. P1*
	b:=a+1;	*wartet, bis die Daten von P0*
		eintreffen
		(implizite) Synchronisation

3 Paralleles Rechnen im shared memory

- Speicher, der gemeinsamen Zugriff der angeschlossenen Prozessoren gestattet
  - exklusiver Zugriff
  - Multiport-Speicher
  - verteilter gemeinsamer Speicher;
    lokale Speicher werden über Netzwerk gekoppelt
    (CTI - Convex Toroidal Interconnect, $\leq 600 MB/s$)
- keine Datenverteilung nötig
- nur noch Synchronisationsoperationen nötig

aber

- Architektur beachten (Cache, Bandbreiten,
  $\rightarrow$ Beispiel: Convex SPP1000)
- Vermeidung von Zugriffskonflikten (meist bieten Compiler schon Mechanismen an)

Beispiel Convex SPP1000:

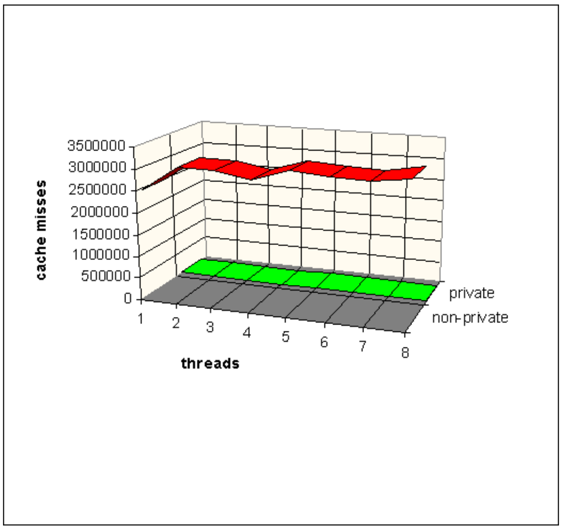
- Compiler kennt verschiedene Direktiven/Pragmas zur Steuerung der Parallelausführung (loop_parallel, prefer_parallel, Tasksteuerung, ordered sections)
- verschiedene Speicherklassen sind möglich (private/shared memory, thread_private, node_private, near_shared, far_shared, block_shared)

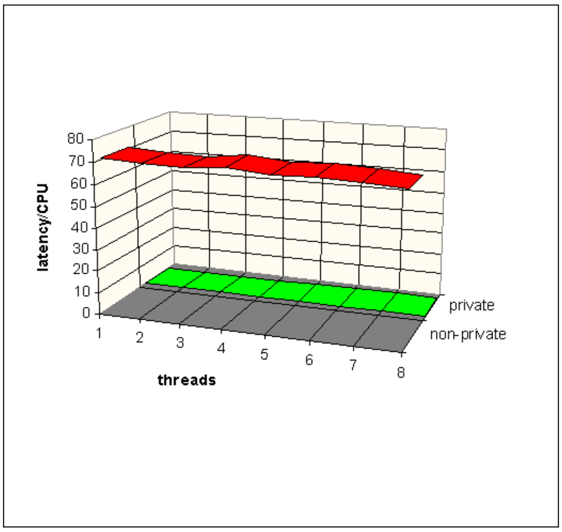
3.1 Beispiel: 1D-FFT auf SPP1000

```
#pragma _CNX loop_parallel(ivar=i), loop_private(j)
for (i=0;i¡n;i++) {
  F[i].real=0;
  F[i].imag=0;
  for (j=0;j<n;j++) {
    F[i].real+=sin((double)(i*j));
    F[i].imag+=cos((double)(i*j));
  }
}
```

CPU time-to-solution:

non-private j	private j
16.291	8.905





3.2 Vermeidung von Zugriffskonflikten

bekannte (Synchronisations)Paradigmen

- Semaphore
- Monitore
- message passing
- CSP - communicating sequential processes

Algorithmen

- Decker Algorithmus
- Algorithmus von Dijkstra
- Algorithmus von Knuth

3.3 dead-locks

Ein dead-lock ist ein Zustand, in dem ein Teilprozeß/Thread unendlich lange auf eine Bedingung oder eine Aktion wartet, die nicht eintritt.

3.3.1 Decker-Algorithmus

```
global t,c1,c2;
t=1;
c1=FALSE;
c2=FALSE;
```

P_1	P_2
c1=TRUE;	c2=TRUE;
while (c2);	while (c1);
if (t==2) {	if (t==1) {
c1=FALSE;	c2=FALSE;
while (t==2);	while (t==1);
c1=TRUE;	c2=TRUE;
}	}
KB;	**KB**;
t=2;	t=1;
c1=FALSE;	c2=FALSE;

3.3.2 Algorithmus von Dijkstra

```
P_i
global t,f[];
local v;
f[i]=PASSIVE;
t=0; do {
  f[i]=DEMAND;
  while (t!=i)
    if (f[t]==PASSIVE) t=i;
  f[i]=BUSY;
  v=0;
  while ((v<n) && ((v==i) || (f[v]!=BUSY))) v++;
} while (v<n);
KB_i;
f[i]=PASSIVE;
```

3.3.3 Algorithmus von Knuth

P_i

```
Test(k) {
  return (∀_{v≠k} (f[v]!=BUSY));
}
global t,f[];
local v;
f[i]=PASSIVE;
t=0;
do {
  f[i]=DEMAND;
  v=t;
  while (v!=i)
    if (f[v]!=PASSIVE)
      v=t;
    else
      v=(v-1) % n;
    f[i]=BUSY;
} while (!(Test(i)));
KB_i;
t=(i-1) % n;
f[i]=PASSIVE;
```

4 Paralleles Objektorientiertes Programmieren

Strategien

- Verteilung von Instanzen (während der Laufzeit, dynamisch)
- a priori Verteilung der Implementation (Client-Server Architektur)
- Verwendung von OO Strukturen zur Beschreibung des Message-Passings (OOMPI)

Ansatz

- template-Klasse mit überladenem ->, *new* und *delete* Operator
- *new* generiert die Instanz und sorgt gemäß einer vorgegebenen Strategie für die Verteilung (Verteilung der Attribute auf Server)
- bei Zugriff auf eine Instanz erhält diese vom Server einen *lock* und eine Kopie wird an den fordernden Prozeß versendet, nach Abarbeitung muß die Kohärenz hergestellt werden
- einheitliches paralleles Programmierschema, d.h. es ist nicht notwendig, einen extra Server zu generieren
- sicherlich nur sinnvoll, wenn **langwierige Operationen** auf **große Datenmengen** angewendet werden

CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture)

- Client-Server-Architektur (Objekt hier **nicht** ausschließlich im Sinne des OO-Paradigmas)
- Objekt-Instanz (Server) ist eine Instanz mit einem gewissen Interface (in IDL beschrieben) diese Instanz wird auf irgendeinem Knoten "gestartet" und bei CORBA registriert (für spätere Aufrufe)
- die Zugriffe auf die Instanz erfolgen über ein inherentes call-by-name, vom Client
- Vererbung ist möglich, läuft aber auch nur wieder auf eine CS-Architektur hinaus (Interfacevererbung!)