Paralleles Rechnen
Einführung

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

31. Mai 1996

*Telefon: 3337, E-Mail: schmei@fz-rossendorf.de

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 Parallelisieren

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;*

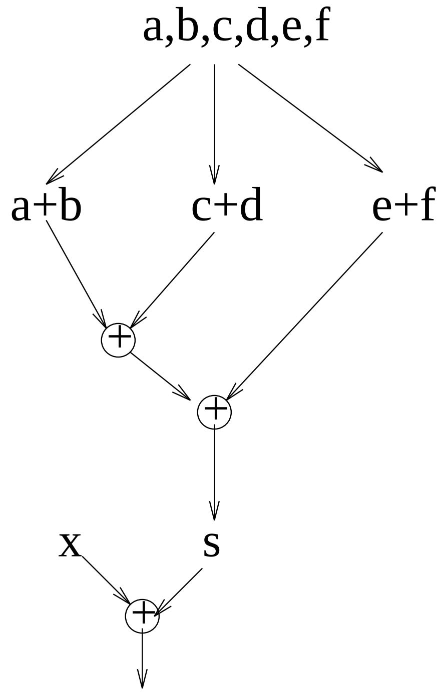
Beispiel für Restrukturierung:

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

→ **scalar replacement, loop fusing, loop unrolling**

3 Parallelisieren

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



$\Rightarrow$ **Datenabhängigkeiten bestimmen**

3.1 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.

3.2 Schleifenparallelisierung

Beispiel:

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

for i:=1 to n do begin
  c[i]:=a[i]+b[i];     { S1 }
  d[i]:=c[i]*a[i];     { S2, S2 datenunabhängig von S1 }
end;

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} δ S_2^i }
end;
```

3.3 Code-Parallelisierung

Beispiel:

- Umstrukturierung des sequentiellen Codes:

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

- Parallelisierung:

P_0	P_1	P_2	P_3
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

4 Kommunikation

- Shared Memory: zwei Prozesse teilen sich einen gemeinsamen Speicher
  (normaler sequentieller Speicher, Multiport-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);*	$\leftarrow$ *Synchronisation nötig*
	b:=a+1;	

- Message Passing: zwischen Prozessen werden Nachrichten verschickt
  (MPI "Message Passing Interface" ist defacto-Standard)

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

5 Paralleles Rechenen auf der Convex SPP1000

- Compiler arbeitet in mehreren Optimierungsstufen:
  - -O0 maschinenunabhängige sequentielle Optimierung
  - -O1 sequentielle Optimierung
  - -O2 O1 + Vektorisierung (C-Serie)
  - -O3 O2 + Schleifenparallelisierung (SPP)

! Der Compiler erzeugt nicht selbstständig parallelen Code aus einer Folge einfacher Anweisungen.

Beispiele und eine Kurzübersicht zum Parallelen Programmieren auf der SPP1000 befinden sich in:
vesta:/home/schmei/parallel/examples bzw.
vesta:/home/schmei/parallel/examples/doc