Paralleles Programmieren auf der **CONVEX SPP1000 Exemplar** (Übersicht)

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

14. Februar 1996

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

1 Wichtige Konstrukte

Grundsätzlich versucht der C-Compiler mit der Option *-O3* eine skalare Optimierung durchzuführen und, wo möglich, Schleifen zu parallelisieren. Die nachfolgend aufgeführten Konstrukte dienen dazu, dem Compiler den Weg zu weisen.
Die Parallelisierung von Iterationen, die nicht als Schleife vorliegen, kann nur per Hand erfolgen.
Sehr wichtig: sollen innerhalb der Applikation *nthread* > 8 **Threads gleichzeitig laufen, so muß das Attribut für das Executable mit** *mpa -m -min nthread -max nthread <filename>* **gesetzt werden.**

1.1 Konstrukte zur sequentiellen Code Optimierung

	#pragma _CNX	Compiler Option
scalar loop	*scalar*	
loop fusing	*prefer_fuse*	-fl, -nfl
loop unrolling	*unroll[(unroll_factor=<n>)]*	-ur, -nur
test promoting		-ptst, -ptstall
scalar replacement		-sr, -nsr
loop carried-dependencies	*no_loop_dependence(<namelist>)*	

1.2 Konstrukte zur direkten Parallelisierung

	#pragma _CNX	Compiler Option
loop parallelization	*loop_parallel[<attributes>]* *prefer_parallel[<attributes>]*	-noautopar
task parallelization	*begin_tasks[<attributes>]* *next_task* *end_tasks*	
critical sections	*critical_section[<gate_t>]* *end_critical_section*	

1.3 Spezielle Funktionen

#include <spp_prog_model.h>

1.3.1 Informationen

Anzahl Prozessoren	*int num_procs()*
Anzahl Threads	*int num_threads()*
Anzahl Hypernodes	*int num_nodes()*
Threads on Node	*int num_node_threads()*
Thread ID	*int my_thread()*
Hypernode ID	*int my_node()*
Parallelization level	*int level_of_parallelism()*

1.3.2 Synchronisation

Gate erzeugen	*int alloc_gate(gate_t*)*
Gate loeschen	*int free_gate(gate_t*)*
Barriere erzeugen	*int alloc_barrier(barrier_t*)*
Barriere loeschen	*int free_barrier(barrier_t*)*
Gate blockieren	*int lock_gate(gate_t*)*
	int cond_lock_gate(gate_t)*
Gate freigeben	*int unlock_gate(gate_t *)*
An Barriere warten	*int wait_barrier(barrier_t*,const int*)*
Synchronisation	*#pragma _CNX SPP sync_routine(<funclist>)*

2 Hinweise zur Parallelisierung

2.1 Parallelisierung von Schleifen

Beispiel:

```
Parallelisierungsanweisung →              #pragma _CNX loop_parallel(ivar=i)
Privatisierung der inneren Laufvariablen → #pragma _CNX loop_private(j)
                                           for (i=0;i<n;i++) {
                                               for (j=0;j<m;j++) {
Code der inneren Schleife →                        ...
                                               }
                                           }
```

Das Privatisieren des inneren Schleifenzählers ist sinnvoll, da sonst nach jedem Durchlauf eine Aktualisierung mit den lokalen Instanzen aller, an der Rechnung beteidigten Threads durchgeführt wird. In diesem Fall ist mit einer sehr hohen

Verzögerung durch Cache-Misses zu rechnen, das heißt, Daten müssen mit einer Latenzzeit von 1.2 bis $2.5 \mu s$ in den Cache nachgeladen werden. Bei nur einem Thread werden dabei etwa etwa 0.01% der CPU-Zeit verbraucht ($2 \rightarrow 45\%$, $3 \rightarrow 65\%$, $4 \rightarrow 70\%$).
Ebenfalls extrem wichtig ist die Privatisierung aller bzgl. eines Threads lokalen Variablen:

static **thread_private** <*typeid*> <*idlist*>

da hier sonst auch wieder bei jedem Zugriff auf eine der Instanzen die Aktualisierung erfolgt. Besonders kritisch sind dabei Datenabhängigkeiten zu betrachten.

Beispiel:

```
void mandel(const double g,const double X1,const double Y2,
  const double xd,const double yd,
  const int maxx,const int maxy,const int maxc,const int z,
  byte **buffer) {

  static thread_private int i,j,k,n,h;
  static thread_private double a,b,m,x,y;

  #pragma _CNX loop_parallel(ivar=i)
  #pragma _CNX loop_private(j,y,x,a,b,m,h)
  for (i=0;i<maxy;i++) {
    y=Y2-i*yd;
    #pragma _CNX no_parallel
    for (j=0;j<maxx;j++) {
      x=X1+j*xd;
      a=0;
      b=0;
      h=0;
      #pragma _CNX no_parallel
      for (n=1;n<=z;n++) {
        m=a*a-b*b+x;
        b=2*a*b+y;
        a=m;
        if (fabs(a)+fabs(b)>g) {
          h=n/2;
          n=z;
        }
      }
      buffer[i][j]=(byte)(h % maxc);
    }
  }
}
```

2.2 Task Parallelisierung

2.2.1 Während der Übersetzung

Um den Compiler explizit anzuweisen, bestimmte Sequenzen parallel laufen zu lassen, werden die Konstrukte *#pragma _CNX begin_tasks*, *#pragma _CNX next_task* und *#pragma _CNX end_tasks* verwendet.

Beispiel (**Datenabhängigkeiten beachten!**):

```
#pragma _CNX begin_tasks
  m=a*a-b*b+x;
#pragma _CNX next_task
  dummy=2*a*b+y;
#pragma _CNX end_tasks
a=m;
b=dummy;
```

2.2.2 Während der Laufzeit

Während der Laufzeit erfolgt die Parallelisierung mit Hilfe der CPS-Bibliotheksfunktionen. Es werden zwei verschiedene Arten der Task(hier =Thread)generierung verwendet; **symmetrische** und **asymmetrische**. Dazu ist zunächst *#include <cps.h>* einzubinden. Der Compiler/Linker muß dann mit der Optionsfolge **-Wl,+parallel,+over,+onenode,+tm,spp1000** und, falls nur ein Hypernode verfügbar ist, zusätzlich mit **+onenode** aufgerufen werden. **Wichtig: an den Aufruf von** *mpa -m -min nthread -max nthread <filename>* **denken.**

symmetrische Erzeugung von Threads:

```
int cps_ppcall(spawn_sym_t*,void (*func)(void*),void* arg)
int cps_ppcalln(spawn_sym_t*,void (*func)(void*),const int*,void*,void*,...)
```

Erzeugt *n* Threads und läßt auf diesen die Funktion *void func(arg[,arg2,...])* laufen. Die Abarbeitung in *main* wird erst fortgesetzt, wenn alle Child-Threads beendet sind.

Beispiel:

```
void fct(int *num) {
  printf("Hello world from %d, %d are running.\n",cps_stid(),*num);
}

int main() {
  spawn_sym_t param;

  param.node=CPS_ANY_NODE;
  param.min =<numthreads>;
  param.max =<numthreads>;
  param.threadscope=CPS_THREAD_PARALLEL || CPS_OVER_SUBSCRIBE;
  cps_ppcall(&param,fct,&(param.min));
  return 0;
}
```

asymmetrische Erzeugung von Threads:

int cps_thread_create(const int,void (*func)(void*),void*)*
int cps_thread_createn(const int,void (*func)(void*),const int*,void*,void*,...)*
Erzeugt einen neuen Thread und übergibt diesem die Funktion *void func(arg[,arg2,...])*. *cps_thread_create* **wartet nicht**, bis *func* beendet ist.

Terminieren eines asymmetrisch erzeugten Threads:

int cps_thread_exit()
Diese Funktion beendet einen Thread explizit, sie wird implizit beim Verlassen, der, bei der Erzeugung übergebenen Funktion gerufen.

Synchronisation bei Beenden asymmetrisch erzeugter Threads:

int cps_thread_wait(const int)*
Falls die Konstante = 1 ist, wartet die Funktion, bis alle Threads beendet sind, sonst liefert sie die Anzahl der noch aktiven Threads.
Achtung! Das Setzen des Flags in einem Child-Thread kann zum Deadlock führen.

Informationen:

stid_t cps_stid()	Thread ID (nur bei symmetrisch erzeugten Threads
ktid_t cps_ktid()	Kernel-Thread ID, irregulär aber "unique"
int cps_nsthreads()	Anzahl der "spawned" Threads ???
int cps_complex_nthreads()	Anzahl Threads in der Applikation
int cps_node_nthreads()	Anzahl Threads auf diesem Hypernode

Synchronisation:

int cps_barrier_alloc(barrier_t)*	Barriereninstanz erzeugen
int cps_barrier_free(barrier_t)*	Barriereninstanz vernichten
int cps_barrier(barrier_t,const int *n)*	Wartet, bis *n* Threads die Barriere erreicht haben
int cps_mutex_alloc(cps_mutex_t)*	Mutex-Instanz erzeugen
int cps_mutex_free(cps_mutex_t)*	Mutex-Instanz vernichten
int cps_mutex_lock(cps_mutex_t)*	Blocking lock der Mutex-Instanz
int cps_mutex_trylock(cps_mutex_t)*	Non-blocking lock der Mutex-Instanz
int cps_mutex_unlock(cps_mutex_t)*	Freigeben der Mutex-Instanz

3 Fehler

1. Exemplar Programming Guide; S. 311 unten; **ktid_t cps_ktid(void);**

2. Examplar Programming Guide; S. 75 oben; #pragma _CNX prefer_fuse wird vom Compiler mit der Meldung "cc: Warning on line <l> of <file>.c: 'prefer_fuse' is not a DIRECTIVE." zurückgewiesen. Verwendet man stattdessen die Compileroption *-fl* so wird ein Loop fusing durchgeführt.

4 Beispiele

Alle Beispiele sind in *vesta:/home/schmei/parallel/examples* zu finden.

1. **Skalare Optimierung**:
   ex1.c

2. **Schleifenparallelisierung**:
   ex2.c, ex3.c, ex4.c

3. **Parallelisierung und kritische Bereiche** (während der Compile-Zeit):
   ex5.c

4. **Parallelisierung und kritische Bereiche** (während der Laufzeit):
   ex6.c

5. **Parallelisierung und Synchronisation** (während der Laufzeit):
   ex7.c