12 POSIX Threads

Portable **O**perating **S**ystem **I**nterface standard for Uni**x**

Was is ein Thread (Aktivitätsfaden)?

Prozess besteht aus

- Adreßraum mit Code globalen Variablen, ...
- Dateien
- Einem oder mehreren Threads

Thread besteht aus

- Stapel
- Registerbelegung

12.1 Überblick

- Thread Erzeugung
- Synchronisationsprimitive:
  - Gegenseitiger Ausschluß
  - Bedingungsvariablen

Hier nicht

- Keys
- Signale
- Scheduling
- Beenden, Warten, Cancel, ... (Warum?)

12.2 Was braucht man

- #include <pthread.h>
- g++ ... -lpthread -lposix4

12.3 Threads Erzeugen

Ein Prozeß hat per default einen Thread, der mit main() beginnt.

```
int pthread_create(
              pthread_t      *new_thread_ID,
        const pthread_attr_t *attr,
              void * (*start_func)(void *),
              void *arg);
```

start_func Womit fängt er an

arg Parameter für ihn

new_thread_ID Zum "Wiederekennen"

attr Wie verhält er sich, was kann er

Parallele Threads:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTE
pthread_create(&tid, &attr, start_func, arg);
```

12.3.1 Semantik

- Globale Variablen gibt nur einmal → impliziet Parameterübergabe über globale Variablen ist gefährlich
- `static` Variablen ebenfalls!
- lokale (`auto`) Variablen sind lokal (stack, register)
- Jeder kann dynamisch Speicher allozieren
- Standard Library ist threadsafe aber es gibt jedes File nur einmal (`stdout` etc!)

12.4 Gegenseitiger Ausschluß

```
     int pthread_mutex_init(pthread_mutex_t *mp,
          const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

`lock` Durch die Tür gehen und hinter sich abschließen oder warten bis jemand rauskommt (britisch in line)

`try_lock` Schauen ob Tür offen ist

`unlock` Aufschließen, rausgehen Tür offen lassen

Beispiel Zähler

```
int sharedCounter = 0;
pthread_mutex_t sharedCounterM;
...initialize somewhere...

int fetchAndIncrement()
{
  pthread_mutex_lock(&sharedCounterM);
  int result = sharedCounter;
  result++;
  sharedCounter = result;
  pthread_mutex_unlock(&sharedCounterM);
  return result;
}
```

Dünnbesetzte Matrizen, Mandelbrotmenge, Master-Worker, ...

12.5 Bedingungsvariablen:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Analogie Schranke an der Unieinfahrt:

`wait` Warte auf Freigabe

`timedwait` mit begrenzter Geduld

`signal` Nächsten durchlassen

`broadcast` Alle durchlassen

Beispiel: Barrier implementieren

12.6 Einsatzzwecke

- Interaktive Programme (nicht hier)
- Asynchrones I/O (nicht hier)
- Primäres Werkzeug zur Parallelisierung (eine shared memory Maschine)
- Lokale Interaktion bei Clustern von shared memory Maschinen
- Entkopplung von Kommunikation und Berechnung Beispiel: Parallele Priority Queue und Branch-and-Bound

12.7 Vergleich mit MPI

+ Speicherzugriff hat geringere Latenz als Kommunikation (Warum?)
+ Eingaben verteilen ist fast geschenkt (Auch schnell?)
+ Speicherzugriff braucht keine Umständlichen Prozeduraufrufe → oft einfacher und flexibler
+ Locks etc. sollten (!?) schneller als T_{start} sein.
– Low Level
  – Aliasing
  – Einfache Lösungen sind oft nicht skalierbar (Beispiel Zähler)
  – Keine kollektiven Operationen
  – Hohe **Bandbreiten** können Tricks erfordern, Cache line alignment, ...
  – Vielfä(lt)(d)ige Fehlermöglichkeiten
– Erfordert spezielle Hardware
– Skalierbarkeit umstritten

12.8 Andere Programmierumgebungen für gemeinsamen Speicher

Programmiersprachen: Concurrent Pascal, Modula-II, Ada, OCCAM (?!)

Libraries: objektorientiert, ...

OPENMP: Compiler `pragmas`. Idee: erhalte Semantik des seq. Programms

Compiler: (halb)automatische Parallelisierung

MPI-2 ?

13 MPI-2

- Einseitige Kommunikation “SM Surrogat”
- Prozeßverwaltung
- Parallele Ein- und Ausgabe
- Kooperation mit Threads
- Erweiterte kollektive Operationen
- C++ Binding
- MPI-3 ?

13.1 Einseitige Kommunikation

- Gegensatz zu “Zweiseitig”
- *Ursprungs*-PE (Sender oder Empfänger) entscheidet *wann* und *wohin* übertragen wird

\+ Schneller (vor allem bei SM) und

\+ Flexibler als zweiseitig

13.1.1 Ein Fenster zur Welt

`MPI_Alloc`: liefert Speicher der besonders geeignet ist. U.u. auch für zweiseitige Kommunikation

`MPI_Win_Create`: Jedes PE in einem Kommunikator macht ein Fenster in seinen Speicher global bekannt. Das

Fenster übernimmt danach die Rolle dieses Kommunikators (für einseitige Kommunikation)

13.1.2 Semantisches Modell

- Private Fensterkopie
- Öffentliche Fensterkopie
- lokale Zugriffe verändern private Kopie
- Auswirkung auf öffentliche Kopie implementierungsabhängig

13.1.3 Get

```
int MPI_Get(
  void *origAdr, int origCnt,
  MPI_Datatype origType,
  int targRank,
  MPI_Aint targDisp, int targCnt,
  MPI_Datatype targType, MPI_Win win);
```

verhält sich ähnlich wie

```
MPI_Irecv(origAdr, origCnt, origType,
          targRank, MPI_ANY_TAG, comm, .....);
```

zusammen mit einem

```
MPI_Isend(
  base + targDisp*disp_unit, targCnt, targType,
  origRank, 42, comm);
```

auf der Gegenseite.

- Zugriff auf öffentliche Kopie von PE `targRank`

13.1.4 Put

- Analog zu Get
- Zugriff auf öffentliche Kopie aber
- Lokale Kopie *kann* beeinflußt werden

13.1.5 Accumulate

- Put mit
- (vordefinierter) Reduktionsoperation (`MPI_op`) z.B. Addition
- MPI garantiert Atomizität bei parallelen Zugriffen

13.1.6 `MPI_Win_fence(assert, win)`

- Kollektiv auf `win`
- Bring alle Operationen auf `win` zum Abschluss
- Lokale Zugriffe werden öffentlich sichtbar
- Put/Get Zugriffe werden lokal sichtbar
- Barrier (wenn `assert`$\neq 0$)

13.1.7 Locks

```
int MPI_Win_lock(int type, int rank, int assert,
                 MPI_Win win);
int MPI_Win_unlock(int rank, MPI_Win win);
```

Lock

- Wie Posix-Thread Mutex auf *ganzes* Fenster `win` von PE `rank`
- Lokale Zugriffe werden öffentlich sichtbar
- `type`: `MPI_LOCK_EXCLUSIVE`, `MPI_LOCK_SHARED`

Unlock

- Freigabe
- Einseitige Kommunikationen auf Ursprung und Ziel beenden
- Abgleich private, öffentliche Kopie

13.1.8 Vergleich mit POSIX Threads

+ Mehr Funktionalität
+ Auch auf distributed Memory
– Umständlich
– In Praxi vermutlich langsamer (Compilerunterstützung?)

13.2 Prozeßverwaltung

13.2.1 `MPI_Comm_Spawn`

- Erzeugt neue Prozesse mit eigenem `MPI_COMM_WORLD`
- Interaktion mit anderen “Welten” über *Interkommunikator*

13.2.2 ...

13.3 Parallele Ein- und Ausgabe

- Files sind Arrays
- Die Sicht (view) eines Prozesses auf eine Datei wird durch einen MPI-Datentyp festgelegt
- Sichten können sich überlappen oder ergänzen
- Blockierender und Nichblockierender Zugriff
- Kollektiver und individueller Zugriff (einschließlich einfacher Form von nichblockierendem kollektiven Zugriff!)
- Shared file pointer erlaubt implizite Master-Worker-Lastverteilung

13.3.1 Fehlt da was?

Tradeoff Implementierbarkeit versus Funktionalität

- Thread-Unterstützung ist optional
- Nichtblockierende kollektive Operationen
- “Active Messages”
- Kanäle und andere persistente Operationen
- Fairness

14 Fehlersuche in Parallelen Programmen

14.1 Eigentlich nichts neues

- Schwer weil Fehlersuche immer schwer.
- Aber nicht viel schwerer: (im Prinzip)
  - Der parallele Anteil ist oft klein
  - SPMD macht Kommunikationsmuster übersichtlich
  - Nachrichtenkopplung macht Interaktion und Synchronisation explizit

14.2 „Langweilige“ Schwierigkeiten

Häufige Programmierfehler

- Werden `MPI_Init` und `MPI_Finalize` richtig verwendet?
- Puffer richtig dimensioniert und übergeben?
- Einigkeit über Typ und Länge einer Nachricht?
- Tags verwechselt
- kollektive Operation *überall* und *konsistent* aufgerufen? (root, ...)

Beim Testen

- Ein/Ausgabe unübersichtlich
- Bei Absturz gehen ggf. Ausgaben verloren!
- Debuggerläufe werden noch schneller unüberschaubar (selbst wenn möglich)

14.3 Defensives Programmieren

- `if` und `switch` vermeiden, ... (siehe auch unten)
- Reichlich Assertions. Vorteil: Keine Belästigung ohne Fehler $\rightsquigarrow$ „Programming by Contract“; Vorbedingungen, Nachbedingungen, Schleifen(in)varianten, Datenstrukturinvarianten
- Kommunikation und Berechnung logisch trennen
- Handler-Routinen zur Behandlung verschiedener Nachrichttentypen ($\rightsquigarrow$ OO-parallele Sprachen, aktive Objekte, Aktor-Sprachen, active Messages) Beispiel: Zufälliges Anfragen
- Lieber zuviele Tags

Der Fall $P = 1$

- Kollektive Operationen OK
- `MPI_ISend`, `MPI_BSend` vermutlich OK.
- `MPI_Send` funktioniert nicht immer.
- `MPI_SSend` kann nicht funktionieren.
- Master-Worker: Master sollte auch Worker sein können.

14.4 Einfache Maßnahmen beim Testen

- Debug-Optionen der MPI-Implementierung aktivieren (ausführliche Fehlermeldungen etc.)
- Alle Assertions aktivieren (routinemäßig)
- Ausgabeströme auseinanderhalten:
  - Mindestens zeilenweise Pufferung (u.U. nicht default bei `stderr`)
  - Ggf. autom. PE-Nummer-Prefix
  - Bei Absturz gehen ggf. Ausgaben verloren!
  - Im Zweifel `MPI_Gather` auf Ausgaben
- Eins, zwei, unendlich
- Landen wirklich alle an den vorgesehenen Synchronisationspunkten (Barrier?)

14.5 „Wissenschaftlich satisfaktionsfähige“ Schwierigkeiten

Deadlock: (zyklisches Warten auf Ressourcen)
$\leadsto$ irgendwas hängt

Lifelock: Nachrichten kommen nicht durch, Terminierung wird nicht festgestellt, . . .
(darum wünscht man sich Fairness)

Race Condition: Fehler treten nur bei bestimmter zeitlicher Abfolge von Ereignissen auf. $\Rightarrow$ schwer reproduzierbar

14.6 Performance Debugging

Modellieren oder Messen

- Anzahl Nachrichten OK?
- Gesamtdatenmenge OK?
- Wartezeiten?
- Flaschenhälse?
- Länge der kritischen Pfade?

Es gibt grafische Werkzeuge mit den gleichen Vor- und Nachteilen wie parallele Debugger. Zusätzlich verfälscht die Messung bei großem P und kleinen Nachrichten die Leistung

15 Ausgewählte Algorithmen für kollektive Kommunikation

15.1 All-to-All für lange Nachrichten

Nachrichten $M_0, \ldots, M_{P-1}$ pro PE

for $j := 1$ **to** $P/2 - 1$ **do**
  for $i := 0$ **to** $P - 1$ **dopar**
    Exchange messages $M_{i+j \bmod P}$ and M_i
    between PEs $i + j \bmod P$ and i

Komplikationen

- Gitter, . . .
- Unregelmäßige Nachrichtenlängen: Üblich Technik ist, Kleinhächseln, an zufällige Adresse, dann erst an Endadresse

15.2 All-Gather für kleine Nachrichtenlängen

[KumarEtAl Program 3.6] Jedes PE hat Nachricht M. PE-Nummer j

$M_{\text{all}} := M$
for $i := 0$ **to** $\log P - 1$ **do**
  $M :=M_{\text{all}}$ from PE $i \oplus 2^j$ (* SendRecv *)
  $M_{\text{all}} :=$ concat(M_{all}, M)

- Interpretation: P verschachtelte Broadcasts
- Kompromisse bei mittleren Nachrichtenlängen
- Analog All-to-all, All-reduce, (All-scan)