Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

Abgabe: 17. November 2009 in der Übung

Übung 9 Goldbach-Theorem mit OpenMP (8 Punkte)

Nach einem unbewiesenen Satz („Goldbach-Theorem“) läßt sich jede gerade Zahl $p \geq 4$ als Summe zweier Primzahlen darstellen (1 wird nicht als Primzahl gezählt). Das lässt sich einfach realisieren:

- Ein Feld $1 \ldots N$ speichert einen `bool`-Wert, ob der Feldindex $i \in 1 \ldots N$ eine Primzahl ist.
- Bei Iteration über das Feld ergibt der Test auf gleichzeitige Primheit für i und $N = i$ dass das Zahlenpaar $(N - i, i)$ die Bedingung erfüllt.

Auf der Homepage finden Sie ein Programm, das für alle Zahlen bis zu einer Obergrenze N die Anzahl der Summen zählt. Darin wird über alle Zahlen $k < N$ iteriert, dann über alle Zahlen $i < k$ und getestet, ob i und $N - i$ prim sind. Parallelisieren Sie das Programm mit *OpenMP* folgendermassen: Die äussere `for`-Schleife (für $k < N$) sollte auf mehrere Threads aufgeteilt werden, Jeder Thread behandelt also einen Teilbereich der Menge $1, \ldots, N$. Da der Rechenaufwand exponentiell steigt, ist die Lastverteilung inhomogen. Untersuchen Sie den Speedup Ihres Programms mit den Scheduling-Parametern `static, guided` und `dynamic` und diskutieren Sie die Ergebnisse.

Es gibt verschiedene andere Optimierungsmöglichkeiten, auch auf algorithmischer Ebene. Zusatzpunkte können Sie erhalten, wenn Sie Strategien zur Optimierung des Problems erläutern (ein Stichwort ist z.B. *Sieb des Eratosthenes*).

Übung 10 Semaphoren (6 Punkte)

Aufbauend auf dem Konzept des Mutex können Semaphoren realisiert werden. Schreiben Sie eine Klasse, die den Datentyp Semaphore implementiert. Verwenden Sie dazu die auf der Vorlesungseite bereitgestellten `ThreadTools`. Diese enthalten allgemeine Werkzeuge zum Handling von Threads, siehe auch die Hinweise am Ende des Aufgabenblattes. Die Klasse `Semaphore` sollte von der Klasse `Condition` abgeleitet werden und folgendes Layout haben:

```
/** Implements a semaphore using Pthreads condition variables */
class Semaphore : private Condition<unsigned int>
{
public:
  //! make a semaphore with initial value
  Semaphore (int init);

  //! make a semaphore with initial value 0
  Semaphore ();

  //! proceed if value
  void P ();

  //! release semaphore
  void V ();
};
```

Sie müssen nur die P- und V-Methoden der Semaphoren implementieren. Dazu benötigen Sie die Methoden `acquire(), wait(), release()` und `signal()` der `Condition`. Die `Condition` enthält ausserdem eine Variable `value`, die die Semaphore erhöhen oder erniedrigen soll. Die bereitgestellten `Threadtools` enthalten bereits eine Datei `semaphore.hh` mit obiger Header-Information sowie zusätzlich in der Datei `semaphore.cc` das Grundgerüst, deren Methoden Sie ausimplementieren müssen. Testen können Sie Ihre Implementierung durch Aufruf des Befehls `make` in der Konsole.

Übung 11 Erzeuger-Verbraucher-Problem mit Semaphoren und Ringpuffer (6 Punkte)

In der Vorlesung haben Sie besprochen, wie Erzeuger-Verbraucher-Probleme mit Semaphoren gelöst werden können. Dieses Problem sollen Sie nun praktisch mit den `ActiveObject`s (deren neuer Name ist `BasicThreads`, siehe die Hinweise am Ende des Blattes) implementieren. Es gebe dabei einen Puffer, der vom Erzeuger gefüllt wird. Ist der Erzeuger am Ende des Puffers angelangt, beschreibt er den Pufferanfang neu, ältere dort gespeicherte Aufträge werden überschrieben. Der Verbraucher liest zu bearbeitende Prozesse vom Pufferende. Für das Programm brauchen Sie

- einen Puffer, der den gewünschten Datentyp speichern kann,
- eine Semaphore, die freie Pufferplätze absichert,
- eine Semaphore, die belegte Pufferplätze absichert.
- Initialisieren Sie die Semaphore, die freie Pufferplätze absichert, zu Beginn mit der Anzahl aller Pufferplätze, da diese initial frei sind.

Schreiben Sie nun Klassen, die Erzeuger und Vebraucher implementieren. Diese sollten wie in der Vorlesung angesprochen von der Klasse `BasicThread` ableiten und müssen die virtuelle Methode `run ()` überladen, in der Erzeuger und Verbaucher ihre Aufgaben durchführen. Die Aufgabe benötigen Sie Ihre Lösung aus der Semaphoren-Aufgabe. Wer diese nicht lösen kann, melde sich bitte per E-Mail beim Tutor.

Testen Sie Ihr Programm mit einer Pufferlänge von 5000. Der Puffer darf beliebige Datentypen speichern. Der Erzeuger soll 100000 Stücke produzieren und der Verbraucher diese konsumieren und gleichzeitig ausgeben, welches Stück er gerade bearbeitet hat.

Hinweise zu den ActiveObjects/ThreadTools:

- Verwenden Sie die in der Vorlesung eingeführten `ActiveObject`s. Diese wurden allerdings nun in `BasicThread`s umbenannt.
- Auf der Vorlesungs-Homepage finden Sie ein zip-Archiv `threadtools.zip`, das die Klassen für die `BasicThread`s, allgemeine Werkzeuge zum Thread-Handling und einige Beispiele bereitstellt. Es ist ein Makefile beigelegt, übersetzen können Sie unter Linux mit `make`. Für andere Betriebssysteme muss eventuell noch etwas Arbeit investiert werden. Für Sie interessant sind die Klassen `Condition` und `BasicThread`.
- Verwenden können Sie alle `ThreadTool`s durch Inkludieren der Datei `tt.hh`. Für die Lösung der Semaphoren-Aufgabe müssen Sie das `Makefile` nicht anpassen, für die Lösung des Erzeuger-Vebraucher-Problems müssen Sie allerdings noch ein passendes Ziel angeben. Orientieren Sie sich dazu am Ziel `PETERSON_APPL` im Makefile. Dieses Beispiel implementiert den Peterson-Lock mit `BasicThread`s. Sie müssen im Prinzip nur ein eigenes Ziel für Ihre Applikation eintragen:

  ```
  MYAPPL          = myapp
  MYAPPL_OBJECTS  = myapp.o basicthread.o semaphore.o condition.o \
     mutex.o barrier.o flag.o
  ...
  all : ... $(MYAPPL)
  ```

  Der `\` markiert einen expliziten Zeilenumbruch. Sie können diesen weglassen, wenn Sie alles in eine Zeile schreiben. Nun müssen Sie noch die Abhängigkeiten für Ihr Ziel angeben:

  ```
  $(MYAPPL) : $(MYAPPL_OBJECTS) Makefile
      $(LINK) $(LLDFLAGS) -o $(EXAMPLE_APPL) $(EXAMPLE_OBJECTS) $(
    LIBS)
  ```

  Achten Sie darauf, dass Sie im `Makefile` niemals Leerzeichen sondern nur Tabs verwenden dürfen. Danach können Sie Ihre Applikation mit `make` übersetzen.