Universität Augsburg
University

Vertiefte Multicore-Programmierung

Universität Augsburg
Institut für Informatik
Prof. Dr. Theo Ungerer
Dr. Sebastian Weis
Dr. Nizar Msadek

03. Mai 2017

Aufgabenblatt 1

Allgemeine Hinweise

Unter `git://ivy.informatik.uni-augsburg.de/git/mc17-${RZ-Kennung}` wurde für jeden Teilnehmer ein Git-Repository angelegt. Sichern Sie Ihre Lösungen bzw. alle Abgaben in diesem Repository. Das jeweilige Übungsblatt und allgemeine Dokumente finden Sie in ihrem Home-Verzeichnis unter *share/multicore/Blattxx*. Quellcodevorlagen erhalten Sie aus dem Git-Repository `git://ivy.informatik.uni-augsburg.de/git/mc17`.
Das Erscheinen zum wöchentlichen Übungsstermin ist verpflichtend. Der Praktikumsraum kann aber auch außerhalb der Übungszeit, sofern dort keine anderen Veranstaltungen stattfinden, genutzt werden. Sorgen Sie außerdem dafür, dass Ihre Abgaben immer (sinnvoll) kommentiert und für Dritte nachvollziehbar sind.

Anleitung zur Benutzung der Git-Repositories

1. Eigenes Git-Repository klonen:
   ```
   git clone ssh://$name@ivy.informatik.uni-augsburg.de:/git/mc17-$name
   cd $name
   ```

2. Remote für Angaben hinzufügen:
   ```
   git remote add assignments $name@ivy.informatik.uni-augsburg.de:/git/mc17
   ```

3. (Neue) Angaben ins eigene Repository übernehmen:
   ```
   git pull assignments master
   ```

Hinweis

Verwenden Sie die vorgegeben Vorlagen für die jeweilige Aufgabe und kompilieren Sie den

Quelltext mit dem vorgegebenen Makefile. Der Standard für alle Aufgaben ist C++17 (`-std=c++17`). Ändern Sie insbesondere nicht die Interfaces der vorgegebenen Klassen, um die Ausführung der mitgelieferten Unit-Tests (`make test`) zu ermöglichen.
Um den Quelltext zu kompilieren, rufen Sie `make` auf. Falls Sie *Eclipse* verwenden sollten, erstellen Sie ein neues Projekt (`File - New - Makefile project with existing code`) und importieren Sie den Ordner der entsprechenden Aufgabe.

1. Aufgabe (1 Punkt)

Implementieren Sie, basierend auf dem Klassen-Rumpf `Counter` in `counter.hpp` eine C++11-Klasse, die einen Zähler inkrementiert und von mehreren Threads aufgerufen werden kann. Das Inkrementieren soll durch Verwendung des `++`-Operators auf dem Objekt realisiert werden, dieser Operator muss dazu überladen werden. Vergleichen Sie den Wert des Zählers bei Programmende mit und ohne Schutz vor gleichzeitigem Zugriff.

2. Aufgabe (2 + 3 Punkte)

Implementieren Sie eine parallele Berechnung der Varianz.

a) Erweitern Sie den Klassen-Rumpf `Barrier` (in `barrier.hpp`) um mit Hilfe eines Mutex und einer Conditional-Variablen eine Barriere zu implementieren. Zur Verwendung der Barriere in mehreren Objekten kann diese statisch instantiiert werden (`static Barrier b(n);` für `n` Threads) oder alternativ im Konstruktor mit übergeben werden. Das Warten an der Barriere wird durch `void Barrier::wait()` realisiert. Nachdem alle Threads `wait()` aufgerufen haben, soll die Barriere automatisch wieder zurückgesetzt werden, sodass eine Wiederverwendung möglich ist.

b) Verwenden Sie oben erstellte Barriere in einem mehrfädigen Programm zur Berechnung der Varianz. In einem ersten Schritt soll der Durchschnitt aller Elemente eines Vektors berechnet werden. Jeder Thread berechnet dazu in einem ihm zugewiesenen Intervall lokal den Durchschnitt. Dieser wird dann mit den Durchschnittswerten der anderen Threads zu einem globalen Durchschnitt verrechnet. Jeder Thread soll danach auf seinem Intervall der Eingabewerte die Summe der quadratischen Abweichungen bilden. Ein Thread akkumuliert letztendlich die lokal berechneten Summen, um die Varianz über alle Eingabewerte zu berechnen. Die Berechnung innerhalb der Threads soll über eine Funktor-Klasse (`Worker` in `worker.hpp`) realisiert werden. Die Kommunikation zwischen den Threads erfolgt über Objekte vom Typ `std::vector`, wobei im Vektor `elements` die Elemente übergeben werden, und die Mittelwerte und Abweichungen in den Vektoren `avg` und `var` zwischengespeichert werden sollen.

3. Aufgabe (4 Punkte)

Entwickeln Sie einen Sudoku-Löser mit Hilfe eines selbst entwickelten Thread-Pools. Dazu werden noch nicht vollständig gelöste Sudokus in einem Stack verwaltet. Ein Thread nimmt dabei jeweils das oberste Rätsel vom Speicher, berechnet alle möglichen Werte für ein noch nicht ausgefülltes Feld und legt Kopien der neuen Teillösungen auf den Stack zurück. Initial befindet sich nur das zu lösende Sudoku auf dem Stack. Der Lösungsalgorithmus soll dabei wie im abgebildeten Pseudocode vorgehen.

Eingabe : Teilweise ausgefülltes Sudoku Feld S

Lege S auf den Stack
for *alle Sudoku Felder S' im Stack* **do**
  Finde die Koordinaten x,y eines unausgefüllten Feldes in S'
  for *alle möglichen Kandidaten k der Koordinate x,y* **do**
    erstelle eine Kopie S' von S
    füge den Wert k an die Stelle x,y ein
    platziere S' zur weiteren Bearbeitung auf dem Stack
  end
end

Implementieren Sie den Sudoku-Löser für Sudokus der Größe 9×9. Stellen Sie sicher, dass das Programm alle möglichen Lösungen findet und danach terminiert. Der Thread-Pool soll eine einstellbare Anzahl von Workern unterstützen und überschüssige Threads in einer Warteschlange verwalten. Testen Sie ihr Programm mit verschiedenen Eingabe-Sudokus. Messen Sie die Laufzeit ihres Programms mit `std::chrono`, untersuchen Sie die Skalierbarkeit ihrer Lösung für 1-32 Worker und versuchen Sie Konflikte beim Zugriff auf den Stack zu reduzieren, indem Sie eine lokale Warteschlange pro Worker-Thread hinzufügen. Sorgen Sie zudem für eine geeignete Last-Verteilung zwischen den Worker-Threads.

Linksammlung

- Git-Getting-Started: `https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`
- Git-Cheat-Sheet: `https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf`
- C++-Tutorial: `http://www.tutorialspoint.com/cplusplus/`
- C++-Tutorial: `http://www.cplusplus.com/doc/tutorial/`

- C++-Referenz: `http://www.cplusplus.com/reference/`
- C++-Referenz: `http://en.cppreference.com/w/cpp`