Aufgaben im Klausurstil zur Vorlesung

High Performance Computing

Dieses Dokument enthält einige Aufgaben im Stil von Klausuraufgaben.
Achtung: die Aufgaben bieten keinen kompletten Überblick über die in der Klausur behandelten Themen und Inhalte. **Klausurrelevant** sind generell **alle in der Vorlesung behandelten Themen**, auch wenn Sie hier in keiner Aufgabe vorkommen! Auch können die zu implementierenden Funktionen für die verschiedenen Parallelisierungsmethoden unterschiedlich sein.

Hinweise:

1. Öffnen Sie das Verzeichnis mit den Aufgaben mit Visual Studio Code. Jede Aufgabe befindet sich in einem eigenen Unterverzeichnis. Bitte bearbeiten Sie die jeweilige Aufgabe im entsprechenden Verzeichnis, indem Sie die bereitgestellten Dateien entsprechend der Anweisungen bearbeiten. Sie können den Code in jedem Verzeichnis mit dem Befehl `make` erstellen und ihn mit dem Befehl `make run` ausführen.
2. Zu jeder Aufgabe existieren eine Reihe von Tests, mit denen Sie Ihren Code überprüfen können. Ein fehlerfreier Durchlauf der Tests ist zu erreichen. Abgaben, die nicht kompilieren können nicht gewertet werden.
3. Wenn eine Aufgabe darin besteht, bestehenden Code zu parallelisieren, muss die Arbeit tatsächlich verteilt werden, das heißt die Berechnung muss echt parallel erfolgen, möglichst gleichmäßig verteilt - es reicht nicht aus, wenn bspw. Prozess oder Thread 0 die ganze Arbeit erledigt.
4. Laden Sie am Ende Ihre Lösung für jede Aufgabe einzeln in Moodle hoch. Laden Sie dazu bitte nur die jeweils relevante .cpp Datei in der entsprechenden Moodle-Aufgabe hoch.
5. Zusätzliche Erläuterung zu den Aufgaben 1-3: Der natürliche Logarithmus von 2 lässt sich aus der sogenannten alternierenden harmonischen Reihe berechnen. Es gilt folgender Zusammenhang:

   $$\ln 2 = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

   Einen Näherungswert erhält man durch Abbruch der Reihe nach m Reihengliedern, diese Berechnung mit m Termen ist in den Aufgaben in der Funktion `ln2(numTerms)` realisiert.

HOCH SCHULE OFFEN BURG

Aufgabe 1: Shared-Memory Parallelisierung **8 Punkte**

Parallelisieren Sie die vorgegebene Funktion `ln2()` zur Berechnung des natürlichen Logarithmus von 2. Verwenden Sie OpenMP für die Parallelisierung auf einem Shared-Memory-System.

Aufgabe 2: Distributed-Memory Parallelisierung **12 Punkte**

Parallelisieren Sie die vorgegebene Funktion ln2() zur Berechnung des natürlichen Logarithmus von 2. Verwenden Sie MPI für die Parallelisierung auf einem Distributed-Memory-System. Teilen Sie dazu die Daten so auf, dass jeder Prozess einen Teil der Arbeit erledigt und führen Sie das Ergebnis am Ende zusammen. Das Ergebnis muss am Ende auf jedem Prozess zur Verfügung stehen, die Tests müssen also auf jedem Prozess erfolgreich sein.
Sie dürfen für Ihre Lösung voraussetzen, dass die Anzahl an Termen `numTerms` durch die Anzahl an Prozessen teilbar ist.

Aufgabe 3: Hybride Parallelisierung **12 Punkte**

Parallelisieren Sie die vorgegebene Funktion ln2() zur Berechnung des natürlichen Logarithmus von 2. Verwenden Sie eine hybride Parallelisierung mit MPI und OpenMP. Das Ergebnis muss am Ende auf jedem Prozess zur Verfügung stehen, die Tests müssen also auf jedem Prozess erfolgreich sein.
Sie dürfen für Ihre Lösung voraussetzen, dass die Anzahl an Termen `numTerms` durch die Anzahl an MPI-Prozessen teilbar ist.

Aufgabe 4: Race-Condition **6 Punkte**

In der Datei min.cpp ist eine Funktion implementiert, die das Minimum einer Funktion in einem Intervall findet. Dabei wird das Intervall in n Zwischenpunkte unterteilt und die Funktion an jedem Zwischenpunkt ausgewertet. Das Minimum aller Funktionsauswertungen wird zurückgegeben.
Die Implementierung verwendet eine Parallelisierung mit OpenMP, um die Suche zu beschleunigen (das ist natürlich insbesondere von Vorteil, wenn die Auswertung der Funktion länger dauert als in den implementierten Tests). Allerdings enthält die Implementierung eine Race-Condition, die in seltenen Fällen das Ergebnis verfälscht.

Finden Sie die Race-Condition und markieren Sie die Code-Zeile mit einem Kommentar.
Beheben Sie die Race-Condition.

Aufgabe 5: Deadlocks **4 Punkte**

Führen Sie das Programm aus - die Ausführung terminiert nicht, da das Programm in einen Deadlock läuft. Beheben Sie den Deadlock, so dass das Programm wie gewünscht läuft (so dass also alle Nachrichten, die Sie im Code vorfinden, korrekt versendet und empfangen werden).

Aufgabe 6: Eigene Datentypen **8 Punkte**

Die Funktion `sendColumn()` realisiert das Senden einer Spalte einer Matrix von Prozess 0 auf Prozess 1. Hierbei wird jeder Eintrag der Matrix einzeln versendet. Bei kleinen Matrizen spielt das noch keine große Rolle, bei großen Matrizen wird das Vorgehen jedoch ineffizient.
Modifizieren Sie die Funktion so, dass für die ganze Spalte nur noch eine einzige Sende- bzw. Empfangsoperation nötig ist.
Tipp: Verwenden Sie einen selbst definierten MPI-Datentyp.