Übungsblatt 4 zur Vorlesung Parallele Programmierung

Abgabe: 25.11.2023, 23:59

Prof. Dr. Michael Kuhn (michael.kuhn@ovgu.de)
Michael Blesel (michael.blesel@ovgu.de)
Parallel Computing and I/O • Institut für Intelligente Kooperierende Systeme
Fakultät für Informatik • Otto-von-Guericke-Universität Magdeburg
https://parcio.ovgu.de

Wir gehen jetzt von unserem seriellen Programm zur Lösung der Poisson-Gleichung aus und betrachten dabei aber nur die Variante des Jacobi-Verfahrens. Hierfür sollen jetzt Parallelisierungen mittels OpenMP erstellt werden. In den Materialien finden Sie eine optimierte serielle Version von partdiff, die für die folgenden Parallelisierungen und das Parallelisierungsschema als Grundlage genutzt werden soll.

Mit der Option `-fopenmp` übersetzt GCC OpenMP-Code. Ein zusätzliches Tutorial zur Programmierung mit OpenMP finden Sie unter folgender URL:

https://hpc-tutorials.llnl.gov/openmp/

1. Parallelisierung mit OpenMP (180 Punkte)

Parallelisieren Sie das Jacobi-Verfahren aus dem seriellen Programm mittels OpenMP. Die parallele Variante muss dasselbe Ergebnis liefern wie die serielle; sowohl der Abbruch nach Iterationszahl als auch der nach Genauigkeit müssen korrekt funktionieren und dieselben Ausgaben wie die seriellen Gegenstücke liefern! Benutzen Sie die OpenMP-Klausel `default(none)`, um Fehler zu vermeiden.

Der erreichbare Speedup des parallelisierten Programms ist abhängig von der konkreten Hardware, mit 1.024 Interlines sollte sich aber bereits ein erkennbarer Speedup ergeben.

Die Threadanzahl soll dabei über den bereits vorhandenen aber bisher ungenutzten ersten Parameter der `partdiff`-Anwendung gesteuert werden können.

Wenn Sie Ihr Programm ausführen, können Sie z. B. mit dem Tool `top` („1“ drücken, um die Cores aufgeschlüsselt zu erhalten) die Auslastung betrachten.

Bitte protokollieren Sie, wieviel Zeit Sie benötigt haben. Wieviel davon für die Fehlersuche?

2. Umsetzung der Datenaufteilungen (60 Punkte)

Zusätzlich zur Standardaufteilung sollen Sie die Daten auf drei verschiedene Arten verteilen:

- Zeilenweise Aufteilung (d. h. Thread 1 bekommt die erste(n) Zeile(n), …)
- Spaltenweise Aufteilung (d. h. Thread 1 bekommt die erste(n) Spalte(n), …)
- Elementweise Aufteilung (d.h. jedes Matrixelement kann von einem anderen Thread berechnet werden)

Implementieren Sie die verschiedenen Datenaufteilungen in Ihrem Programm, wobei für jede Datenaufteilung eine separate `calculate`-Funktion genutzt werden soll. Wenn verschiedene Datenaufteilungen parallelisiert sind, geben Sie den Code bitte so ab, dass die Binärdateien für die entsprechenden Datenaufteilungen jeweils ein Target sind. Hierzu kann beim Kompilieren `-D` verwendet werden, um Flags während der Kompilierung zu setzen. Die mit `-D` definierten Makros können dann wie folgt im Code benutzt werden:

```
#ifdef ELEMENT
...
#endif
```

In den Materialien finden Sie ein bereits modifiziertes `Makefile` und eine `partdiff.c` mit vier `ifdef`-Klauseln (`DEFAULT` für Aufgabe 1 und drei weitere Fälle für Aufgabe 2).

Hinweis: Für diese Aufgabe muss eventuell der Code der Schleifen umgeschrieben werden.

3. Leistungsanalyse (180 Punkte)

In dieser Aufgabe soll die Leistungsfähigkeit der parallelisierten Variante analysiert werden. Das parallelisierte Programm sollte bei Ausführung mit 24 Threads und 4.096 Interlines einen Speedup von mindestens 10 erreichen.

Wiederholen Sie dabei jede Messung mindestens drei Mal, um aussagekräftige Mittelwerte bilden zu können. Dafür können Sie beispielsweise das Werkzeug `hyperfine` benutzen, das Sie mit `module load hyperfine` laden können. Es ist außerdem empfehlenswert die Störfunktion $f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y)$ zu verwenden, da der erhöhte Rechenaufwand das Skalierungsverhalten verbessert.

Für `t` Threads, `i` Interlines und `n` Iterationen können Sie folgenden Aufruf benutzen:

```
./partdiff t 2 i 2 2 n
```

Die Messungen sollen mit Hilfe von SLURM auf einem der Rechenknoten durchgeführt werden. Geben Sie die für die Messungen verwendete Hardwarekonfiguration (Prozessor, Kernanzahl, Arbeitsspeichergröße etc.) an und nutzen Sie den gleichen Rechenknoten für eine Messreihe.

In den Materialien finden Sie im `slurm`-Verzeichnis vorgefertigte Job-Skripte, welche Sie für die Messreihen verwenden können. Zum Ausführen sollten Sie sich im `slurm`-Verzeichnis befinden und von dort aus die Skripte mit `sbatch` starten:

```
$ cd PP-2023-Uebung-04-Materialien
$ make -C pde
$ cd slurm
$ sbatch messung1.slurm
$ sbatch messung2.slurm
```

Messung 1

Ermitteln Sie die Leistungsdaten Ihres OpenMP-Programms und vergleichen Sie die Laufzeiten für jeweils 1–24 Threads in einem Diagramm. Vergleichen Sie Ihre Variante auch unbedingt mit dem ursprünglichen seriellen Programm! Verwenden Sie hierzu 4.096 Interlines. Der kürzeste Lauf sollte mindestens 30 Sekunden rechnen; wählen Sie geeignete Parameter aus!

Führen Sie diese Messung sowohl für die Standardaufteilung als auch für die zusätzlichen Datenaufteilungen aus Aufgabe 2 durch. Welchen Grund kann es für die gemessenen Ergebnisse geben? Schreiben Sie dazu ca. eine Seite Erklärungen.

Messung 2

Ermitteln Sie weiterhin, wie Ihr OpenMP-Programm in Abhängigkeit der Matrixgröße (Interlines) skaliert. Verwenden Sie hierzu 24 Threads und 13 Messungen zwischen 1 und 4.096 Interlines (wobei Interlines = 2^i für $0 \leq i \leq 12$). Der längste Lauf sollte maximal 30 Minuten rechnen. Starten Sie mit 4.096 Interlines und wählen Sie geeignete Parameter aus; für die folgenden Läufe können Sie die Interlines dann entsprechend der gegebenen Formel verringern. Visualisieren Sie alle Ergebnisse in hinreichend beschrifteten Diagrammen.

Führen Sie diese Messung nur für die Standardaufteilung durch. Schreiben Sie ca. eine halbe Seite Interpretation zu diesen Ergebnissen.

4. Parallelisierungsschema (180 Punkte)

Ziel der kommenden Übungsblätter wird sein, die partdiff-Anwendung mittels POSIX Threads und MPI zu parallelisieren. Zunächst soll ein Parallelisierungsschema erstellt werden.

Gehen Sie so vor, dass Sie die gesamten Matrixdaten auf die Threads bzw. Prozesse (im Folgenden: Tasks) aufteilen, d. h. jeder Task bearbeitet einen Teil der Daten. Beachten Sie dabei folgendes: Zur Berechnung der Werte einer Zeile benötigt man immer die Werte der darüber- und der darunterliegenden Zeile (siehe folgende Grafik).

···	`Matrix[i - 1][j]`	···
`Matrix[i][j - 1]`	`Matrix[i][j]`	`Matrix[i][j + 1]`
···	`Matrix[i + 1][j]`	···

Wenn also die zu berechnende Zeile die erste oder letzte des Blockes eines Tasks ist, so wird die benötigte Nachbarzeile von einem Nachbartask verwaltet; dieser muss sie dann unter Umständen weitergeben. Gleichermaßen muss man, wenn man eine Randzeile neu berechnet hat, sie unter Umständen an den entsprechenden Nachbartask weitergeben. Die Problematik liegt darin, dies zum richtigen Zeitpunkt mit den richtigen Werten zu tun. Je nachdem, ob auf gemeinsamem oder verteiltem Speicher gearbeitet wird, unterscheidet sich die Weitergabe der Randzeilen. Überlegen Sie sich außerdem, welche Tasks auf welche Variablen zugreifen müssen und ob dadurch Konflikte entstehen können.

Beachten Sie auch, dass der Berechnungsablauf bei den beiden Verfahren (Jacobi und Gauß-Seidel) unterschiedlich ist und somit auch das Kommunikationsschema unterschiedlich ausfallen wird. Beachten Sie weiterhin, dass der erste und der letzte Task keinen vorhergehenden bzw. nachfolgenden Nachbartask mehr haben.

Um das Speicherverhalten zu optimieren, dürfen die Tasks bei verteiltem Speicher nur die benötigte Teilmatrix im Speicher halten. Somit können auch Probleme berechnet werden, welche nicht in den Hauptspeicher eines einzelnen Knotens passen.

Gehen Sie auch auf Probleme ein, die sie bei der Beendigung des Programms feststellen. Geben Sie für jeden der Fälle eine kurze Beschreibung an, welche Probleme auftreten könnten, und welche Mittel Sie zu deren Lösung vorschlagen würden (falls eine Lösung möglich scheint).

Diskutieren Sie dabei mindestens die folgende Punkte ausführlich.

1. Beschreibung der Datenaufteilung der Matrix auf die einzelnen Tasks
    - Welche Daten der Matrix werden von welchem Task verwaltet?
    - Visualisieren Sie die Datenaufteilung mit geeigneten Grafiken.
2. Parallelisierungsschema für das Jacobi-Verfahren
    - Beschreiben Sie aus Sicht eines Tasks, wann die Berechnung und wann die Kommunikation mit seinen Nachbarn erfolgt. Unterscheiden Sie nach gemeinsamem und verteiltem Speicher.
    - Welche Daten benötigt der Task von seinen Nachbarn und wann tauscht er die Daten aus?
    - Auf welche Variablen bzw. Daten muss welcher Task zugreifen?
3. Parallelisierungsschema für das Gauß-Seidel-Verfahren (siehe Jacobi)
4. Diskussion der Abbruchproblematik
    - Es sind vier Fälle zu betrachten: Abbruch nach Iterationszahl und Genauigkeit für jeweils Jacobi und Gauß-Seidel.
    - Wann wird ein Task feststellen, dass das Abbruchkriterium erreicht wurde und er seine Arbeit beenden kann?
    - In welcher Iteration befindet sich ein Task im Vergleich zu seinen Nachbarn, wenn er das Abbruchkriterium erreicht?

Abgabe

Als Abgabe werten wir den letzten Commit vor der Abgabefrist in Ihrem Git-Repository. Im Hauptverzeichnis des Repositories wird ein Verzeichnis `PP-2023-Uebung-04-Materialien` mit folgendem Inhalt erwartet:

- Eine Datei `gruppe.txt` mit den Gruppenmitgliedern (eines je Zeile) im folgenden Format:

    ```
    Erika Musterfrau <erika.musterfrau@example.com>
    ```

```
Max Mustermann <max.mustermann@example.com>
```

- Der überarbeitete Code des `partdiff`-Programms im Verzeichnis `pde` (Aufgaben 1–2)
  - Ein `Makefile` mit Targets `partdiff-{element,spalte,zeile}` für Binärdateien `partdiff-{element,spalte,zeile}`, welche jeweils die entsprechende Datenaufteilung umsetzen (Aufgabe 2)
- Eine Ausarbeitung `leistungsanalyse.pdf` mit den ermittelten Laufzeiten und der Leistungsanalyse (Aufgabe 3)
- Eine Datei `parallelisierungsschema.pdf` mit Ihren Antworten (Aufgabe 4)