Übungsblatt 3 zur Vorlesung Parallele Programmierung

Abgabe: 09.11.2024, 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

---

1. SLURM-Nutzung (20 Bonuspunkte)

Nach dem Login via SSH befinden Sie sich auf dem sogenannten Login-Knoten des Clusters. Dieser ist primär zum Verwalten Ihrer Daten und Kompilieren von Software gedacht. Aufwändigere Aufgaben sollten immer auf Rechenknoten durchgeführt werden.

Der Cluster nutzt SLURM zur Job-Verwaltung, welches genutzt werden kann, um Zugriff auf die Rechenknoten zu erlangen. Grundsätzlich ist dabei zwischen nicht-interaktiver und interaktiver Nutzung zu unterscheiden. Im nicht-interaktiven Fall schicken Sie sogenannte Jobs ab, die SLURM auf den Rechenknoten ausführt. Im interaktiven Fall reservieren Sie sich einen Rechenknoten und loggen sich auf diesem ein.

Alle verfügbaren Partitionen und Knoten können Sie mit `sinfo` anzeigen. Die für die Vorlesung bzw. Übungen genutzte Partition heißt `vl-parcio`. Zur interaktiven Nutzung können Knoten mit `salloc` reserviert werden. `salloc` startet dabei eine Subshell, die die Reservierung aufrecht erhält bis Sie die Subshell mit `exit` verlassen.

```
$ salloc -p vl-parcio
$ srun --pty bash
```

Die beiden Befehle können für eine temporäre Allokation auch zusammengefasst werden.

```
$ srun -p vl-parcio --pty bash
```

Standardmäßig wird ein einzelner Kern (mit zwei Threads) allokiert. Zusätzliche Kerne lassen sich mit dem Parameter `-c` anfordern, wobei die Anzahl an Threads angegeben werden muss. Ein Wert von 4 entspricht beispielsweise zwei Kernen mit jeweils zwei Threads. Die allokierte Kern- bzw. Threadanzahl lässt sich einfach mit dem Kommando `nproc` kontrollieren.

```
$ srun -p vl-parcio -c 4 nproc
```

Frage: Wieso werden vier Threads allokiert, wenn Sie mit `-c` drei anfordern? Schreiben Sie dazu einige Sätze Begründung.

Einen nicht-interaktiven Job können Sie mithilfe von `sbatch job.slurm` abschicken. Aktive Jobs können mit `squeue` angezeigt werden; die Ausgabe kann mit `squeue -u <user>` auch auf eigene Jobs eingeschränkt werden. Wenn notwendig, kann ein Job auch mit `scancel <job-id>` abgebrochen werden. Das folgende Job-Script kann als Vorlage genutzt werden.

```
#!/bin/bash

# Time limit is ten minutes (see "man sbatch")
#SBATCH --time=10:00
# Run one task on one node
#SBATCH --nodes=1
#SBATCH --ntasks=1
# Make all cores available to our task
#SBATCH --cpus-per-task=48
# Use lecture partition
#SBATCH --partition=vl-parcio
# Redirect output and error output
#SBATCH --output=job.out
#SBATCH --error=job.err

srun hostname
srun nproc
srun ./my-application
```

Standardmäßig wird ein Task auf einem Knoten allokiert. Die Anzahl an Tasks lässt sich mit dem Parameter `-n` (oder `--ntasks`) anpassen, die Anzahl an Knoten mit dem Parameter `-N` (oder `--nodes`). Beide Parameter werden erst bei der Programmierung mit MPI notwendig.

2. Paralleles Starten eines Shell-Scripts (40 Punkte)

1. Erstellen Sie ein Shell-Script `timescript.sh` welches folgende Ausgabe erzeugt:

   `HOSTNAME: TIMESTAMP`

   HOSTNAME: Kurzer Hostname des Rechners, auf dem das Script ausgeführt wird (Tipp: `hostname --short`).

   TIMESTAMP: Zeitstempel zur Zeit der Ausführung des Scripts in einem mindestens auf die Mikrosekunde genauen Format (Tipp: `date --iso-8601=ns`).

   (Tipp: Sehen Sie sich die Manpages von `hostname` und `date` an.)

2. Erstellen Sie ein Job-Script `jobscript.sh`, das `timescript.sh` gleichzeitig auf vier Knoten mit je vier Prozessen startet. Dabei soll als Ausgabe eine Datei `timescript.out` entstehen, die die Ausgabe von *jedem* Aufruf von `timescript.sh` beinhaltet. Das Script soll mit `sbatch jobscript.sh` aus der Shell aufgerufen werden können.
   (Tipp: Beschäftigen Sie sich mit `srun`, `#SBATCH -N` und `#SBATCH -n`.)

3. Nachdem `jobscript.sh` die Datei `timescript.out` geschrieben hat, soll es „fertig“ ausgeben. Modifizieren Sie das Script so, dass dieses „fertig“ in einer Datei mit dem Namen `jobscript.out` steht.

4. Führen Sie das Script mehrmals aus.

- **Frage:** Was fällt Ihnen an den Zeitstempeln auf? Versuchen Sie Ihre Beobachtung zu erklären!
- **Frage:** Könnte man die Datei `timescript.out` auch innerhalb des Scriptes `timescript.sh` erzeugen? Falls ja: Wie? Falls nein: Warum nicht?

3. Leistungsoptimierung (250 Punkte)

Mittels iterativer Verfahren soll die Poisson-Gleichung für folgende Fälle gelöst werden:

Wir versetzen Sie jetzt in die typische Situation vieler Diplomanden und Doktoranden, die das Gebiet der parallelen Programmierung betreten: Gegeben ist der Code eines seriellen Programms. Er hat eine siebenstellige Zahl von Code-Zeilen, ist in einer Ihnen nicht bekannten Programmiersprache erstellt, außerdem ist er unkommentiert und der ursprüngliche Programmierer ist bereits verstorben. Immerhin aber haben Sie die Quellen (siehe Materialienseite).

Die Quellen beinhalten den kompletten Quelltext, einige weitere Informationen und ein unter Linux ausführbares Programm namens `partdiff`. Verwenden Sie dieses zum Testen des Verfahrens. Spielen Sie ein bisschen mit den Parametern und sehen Sie sich die beigelegten Referenzlösungen an.

Die Problemgröße N wird mittels der Variable `interlines` nach der Formel $N = 9 + (8 \times \texttt{interlines}) - 1$ berechnet. Dadurch wird die Ausgabe mit der Funktion `DisplayMatrix()` überschaubar, die immer genau 9 Zeilen und Spalten ausgibt. Verwenden Sie diese Ausgabe, um die Richtigkeit Ihrer Änderungen zu überprüfen.

Im seriellen Code sind zwei Algorithmen zur Lösung des angeführten Poissonproblems implementiert (Abschnitt 4.4): Das Jacobi-Verfahren und das Gauß-Seidel-Verfahren. Ein Überblick über die mathematischen Grundlagen ist im Anhang gegeben.

In diesem Code sind einige Leistungsprobleme vorhanden, die Sie finden und beheben sollen. Das Programm sollte nach Ihren Änderungen ungefähr um einen Faktor von 10 schneller laufen. Gehen Sie dabei wie folgt vor, wobei die Auswirkung jeder Änderung auf die Laufzeit gemessen werden soll:

1. Profilen Sie das Programm mit Hilfe von `gprof`, um zeitintensive Funktionen zu identifizieren und zu optimieren.
2. Überprüfen und optimieren Sie die Speicherzugriffsmuster.
3. Kompilieren Sie das Programm mit verschiedenen Compiler-Optimierungen.

Ihre Änderungen sollten sich zum Großteil auf die `calculate`-Funktion beschränken. Nehmen Sie keine Optimierungen vor, die automatisch vom Compiler durchgeführt werden (z. B. Elimination überflüssiger Variablen, Umsortierung von Codezeilen).

Um `gprof` benutzen zu können, passen Sie das `Makefile` an, um `partdiff` mit der `-pg`-Option zu kompilieren und zu linken. Lassen Sie danach das Programm laufen, wird eine Datei `gmon.out` geschrieben. Rufen Sie jetzt `gprof ./partdiff` auf, um sich die aufgezeichneten Daten anzusehen. Erläutern Sie die Ausgabe.

Dokumentieren Sie Ihre Änderungen und messen Sie für jede Änderung die Verbesserung in der Programmlaufzeit. Die Programmlaufzeit können Sie mit Hilfe von `time` bestimmen, indem Sie einfach `time ./partdiff ...` ausführen. Führen Sie `partdiff` dabei wie folgt aus:

```
./partdiff 1 2 64 2  5120
```

Wichtig: Die Ergebnisse müssen vor und nach Ihren Änderungen identisch sein! Im Unterverzeichnis `referenz` der partdiff-Anwendung finden Sie dafür auch Referenzergebnisse.

3.1. Mathematische Optimierung (30 Bonuspunkte)

Denken Sie darüber nach ob die mathematischen Operationen im Code in einer effizienteren Weise durchgeführt werden können. Sie müssen dafür nicht die zugrundeliegenden Algorithmen verändern. (Hinweis: Schauen Sie sich die mathematischen Operationen an und überlegen Sie ob es einen schnelleren Weg gibt diese auszuführen)

Abgabe

Als Abgabe werten wir den letzten Commit vor der Abgabefrist in Ihrem Git-Repository. Im Hauptverzeichnis des Repositories wird ein Verzeichnis `PP-2024-Uebung-03-Materials` mit folgendem Inhalt erwartet:

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

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

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

- Eine Datei `slurm-antworten.md` mit Ihren Antworten (Aufgabe 1)
- Eine Datei `timescript-antworten.md` mit Ihren Antworten, die auf dem Cluster ausführbaren Scripte `timescript.sh` und `jobscript.sh` sowie eine Datei `timescript.out` mit der Ausgabe eines Durchlaufs Ihres Scriptes mit mehreren Prozessen (Aufgabe 2)
- Eine Datei `pde-optimierung.md` mit Ihren Ergebnissen (Aufgabe 3)
  - Beschreiben Sie die von Ihnen vorgenommenen Optimierungen und die dadurch erreichten Leistungsgewinne
  - Ausgabe von `gprof` mit Erläuterungen.
  - Der überarbeitete Code des `partdiff`-Programms im Unterverzeichnis `pde`

4. Mathematischer Hintergrund

Hinweis: Dieser Teil soll Interessierten dabei helfen, den Hintergrund des Programms zu verstehen. Falls Sie Probleme haben, den Ablauf des Programms zu verstehen, kann die Erklärung des Verfahrens im folgenden Text hilfreich sein. Es ist für die Bearbeitung der Aufgaben allerdings nicht notwendig die kompletten mathematischen Hintergründe zu verstehen.

Viele natürliche und technische Vorgänge lassen sich durch partielle Differentialgleichungen beschreiben. Ein Beispiel hierfür ist die Poisson-Gleichung. Mangels vorhandener analytischer Lösungsformeln muss man sich oft Methoden der numerischen Mathematik bedienen.

Hier gelangt man zunächst durch

1. Diskretisierung (Festlegung der interessanten Punkte im gewünschten Lösungsgebiet)
2. Ersetzen der Differentialquotienten durch Differenzenquotienten

zu einem System von linearen Gleichungen.

Für die Berechnung des Lösungsvektors dieses Systems existieren direkte und indirekte Verfahren. Wegen der Nachteile der direkten Verfahren (Eliminationsverfahren wie z. B. die Gauß-Elimination), nämlich zu hohe algorithmische Komplexität einerseits und numerische Instabilität andererseits, bevorzugt man heute indirekte Verfahren, zum Beispiel Iterationsverfahren, bei denen man sich iterativ bis zu einer gewünschten Genauigkeit der exakten Lösung annähern kann (sofern das Iterationsverfahren konvergiert). Zwei dieser Iterationsverfahren werden hier kurz und pragmatisch vorgestellt.

4.1. Problemstellung

Gegeben ist eine partielle Differentialgleichung der Form

$$-u_{xx}(x,y) - u_{yy}(x,y) = f(x,y) \text{ mit } 0 < x, y < 1 \tag{1}$$

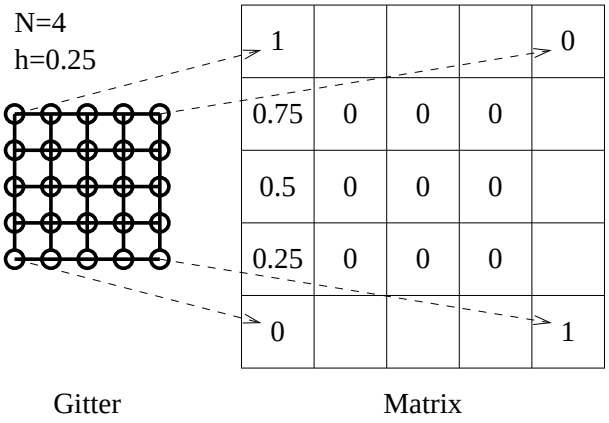
Diese Darstellung wird als Poisson-Problem bezeichnet. u_{ii} ist die zweite Ableitung der Funktion u nach i. Die Funktion $f(x,y)$ bezeichnet man als Störfunktion. Die Randwerte $u(\_,0)$, $u(\_,1)$, $u(0,\_)$ und $u(1,\_)$ sind gegeben. Gesucht wird $u(x,y)$ für $0 < x, y < 1$.

4.2. Diskretisierung

Die Lösung der Poisson-Gleichung soll auf dem Gebiet $[0,1] \times [0,1]$ berechnet werden. Einfachste Diskretisierung ist ein äquidistantes quadratisches Gitter.

Anzahl Intervalle in jeder Richtung: N
Anzahl Punkte auf Gesamtgebiet (mit Rand): $(N+1)^2$
Anzahl innerer Punkte: $(N-1)^2$
Gitterweite h: $1/N$

Dieses Gitter kann in einer $(N+1) \times (N+1)$-Matrix gespeichert werden. Jeder Eintrag in der Matrix repräsentiert einen Punkt des Gitters. Die Randpunkte des Gitters werden vor Beginn der Berechnung vorbelegt.



4.3. Übergang von Differential- zu Differenzenquotienten

Wir definieren die inneren Gitterpunkte

$$u_{i,j} := u(i * h, j * h) \text{ mit } i, j = 1, ..., N - 1 \tag{2}$$

Die Ersetzung der partiellen Ableitungen aus (1) durch finite Differenzen zweiter Ordnung:

$$u_{xx;i,j} := 1/h^2(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})$$

$$u_{yy;i,j} := 1/h^2(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})$$

liefert ein System von linearen Gleichungen der Form:

$$1/h^2(4v(i,j) - v(i-1,j) - v(i+1,j) - v(i,j-1) - v(i,j+1)) = f(i,j) \tag{3}$$

Vorgehensweise zur Lösung des Systems linearer Gleichungen

Darstellung von (3) in Matrixform: $Au = h^2 f$ (Herleitung und Aufbau der Matrix A ist unbedeutsam für die Anwendung des Iterationsverfahrens.)

Exakte Lösung: u
Näherung: v (soll iterativ berechnet werden, bis v nur noch minimal von u abweicht)
Fehler: $e = u - v$
Leider: u ist unbekannt, d.h. e ist nicht feststellbar.
Aber: berechenbar sind

- Residuum $r := h^2 f - Av$ (r ist der Betrag, um den die Näherung von v vom Originalproblem $Au = h^2 f$ entfernt ist)
- Norm des Residuums $\| r \|_\infty := \max | r(i,j) |$
- Zusammenhang: $\| r \|_\infty = 0 \iff e = 0$

Aus $Au = h^2 f$ und $Av = h^2 f - r$ erhält man durch Subtraktion $Ae = r$, bzw. $e = A^{-1} r$, genannt Residuumsgleichung.

Zum Berechnen einer Näherung $\hat{e}$ für e: Verwende in der Residuumsgleichung statt A die einfach zu invertierende Matrix $D = (d_{i,j})$ mit $d_{i,j} = 4\delta(i, j)$ (δ: Kronecker-Symbol).

D ist die Matrix, die aus A durch Nullsetzen sämtlicher Nicht-Hauptdiagonalelemente hervorgeht.

Vorgehensweise:

1. Initiales v^0 berechnen oder raten (einfachheitshalber gleich 0 setzen). Setze $i = 0$.
2. Setze $i = i + 1$. Berechne r^i mittels $r^i = h^2 f - Av^{i-1}$.
3. Berechne $\hat{e}$ mittels $\hat{e} = D^{-1}r$.
4. Berechne neue Näherung $v^i = v^{i-1} + \hat{e}$.
5. Wenn $\| r \|_\infty < Schranke$, Abbruch, sonst zu 2.

4.4. Iterative Lösungsmethoden

Prinzip: Erste Näherung raten, dann iterativ verbessern.

Jacobi-Verfahren: Verwendet zwei Matrizen für v: Die Aktualisierung der neuen Werte erfolgt durch Betrachtung der alten Werte.

Gauß-Seidel-Verfahren: Verwendet nur eine Matrix für v, d.h. neue Werte werden verwendet, sobald sie berechnet wurden.

4.5. Pragmatische Vorgehensweise

Schema für das Programm zur Lösung der Poisson-Gleichung:

```
Initialisiere Matrix (Ränder und innere Punkte)
Solange (Maximum_Residuum > Schranke)
    Über alle Zeilen DO
      Über alle Spalten DO
         // Berechne den Abtaststern
         star =                    - v[m_old][x][y+1]
               - v[m_old][x-1][y] + 4*v[m_old][x][y] - v[m_old][x+1][y]
                                   - v[m_old][x][y-1];
         // Berechne den Korrekturwert
         korrektur = (f(h*x,h*y) * h_square - star) / 4;
         // Für Abbruchbedingung
         Berechne Norm von Residuum
         Berechne Maximum_Residuum
         // Neue Belegung der Matrix
         v[m_new][x][y] = v[m_old][x][y] + korrektur;
         // Gauß-Seidel: m_new = m_old
```

Literatur (begleitend und weiterführend)

- Stoer, Bulirsch: Einführung in die numerische Mathematik II, Heidelberger Taschenbücher, Band 114, Springer
- William H. Press: Numerical Recipes in Pascal/C/Fortran, Cambridge, USA 1990