
Aufgabe 2 muss erst in zwei Wochen abgegeben werden, damit Sie genug Zeit für die Programmieraufgabe haben.

– Das HEAS-Team

1 Fragen zur Vorlesung (30 Punkte)

1.1 MPI-IO-Implementierung (26 Punkte)

1. Welche Zugriffsarten gibt es auf die Daten einer Datei vom Standpunkt der Optimierung des parallelen Zugriffs?
2. Welche dieser Zugriffsarten erlauben dem darunterliegenden parallelen Dateisystem die größten Optimierungen?
3. Wie funktioniert Data Sieving beim Lesen?
4. Wie funktioniert Data Sieving beim Schreiben?
5. Welche Probleme gibt es bei Data Sieving?
6. Wie funktioniert Two-Phase I/O?
7. Welche MPI-IO-Hints kontrollieren bei ROMIO Data Sieving?
8. Welche MPI-IO-Hints kontrollieren bei ROMIO Two-Phase I/O?
9. Welche MPI-IO-Hints kontrollieren PVFS?

1.2 Metadatentests (4 Punkte)

1. Welche Metadatenoperationen haben wir getestet?
2. Was für einen Einfluss hat die Anzahl der Unterverzeichnisse auf die Metadatenleistung von Dateisystemen?

2 Einbauen von I/O in ein bestehendes paralleles Programm (600 Punkte)

In *Cluster Computing I* haben Sie den sequentiellen Code für das Jacobi-Verfahren (und auch Gauß-Seidel) parallelisiert. Aufgrund von Umwelteinflüssen und Verschleißerscheinungen der Hardware kann es immer wieder zu Ausfällen bei der Ausführung des Programmes kommen, welche einen Abbruch der Berechnung zur Folge haben. Um zu vermeiden, dass die ganze Anwendung ihre Berechnung von vorne beginnen muss, soll nun von Zeit zu Zeit (alle C Iterationen; über die Kommandozeile steuerbar) der aktuelle Rechnungszustand in eine Datei gesichert werden (Checkpointing). In dieser Datei soll zunächst einfach nur der aktuelle Stand der Matrix gespeichert werden (ohne Metainformationen). Das Wiederaufnehmen der Berechnung muss von Ihnen nicht implementiert werden.

Weiterhin will ein Mathematiker die Genauigkeit der Integration über den Lauf des Programmes verfolgen. Hierfür soll die Diagonale der Matrix alle V Iterationen in eine Datei *visualization.dat* angehängt werden. Ein weiteres (bereits fertiggestelltes) Visualisierungswerkzeug `partdiff-vis` kann auf diese Datei angewendet werden und gibt online den Fortschritt der Integration aus.

Die zu Visualisierungszwecken verwendete Datei soll hierbei den folgenden Header mit der angegebenen semantischen Bedeutung aufweisen:

4-Byte-Integer - Entspricht der Größe der Matrix (N+1)

4-Byte-Integer - Entspricht der Länge des folgenden lesbaren Strings

Lesbarer String - Inhalt: `Processes: <Prozessanzahl>, Matrix length: <N+1>`

Für jede zu visualisierende Iteration wird nun folgende Information angehängt:

4-Byte-Integer - Aktuelle Iterationsnummer

Double[N+1] - Diagonale der Matrix

Sobald der Jacobi-Integrator seine Aufgabe beendet hat, wird noch der *4-Byte-Integer* `-1` an die Datei angehängt, um dem Visualisierungswerkzeug zu signalisieren, dass die Integration nun abgeschlossen ist.

Eine Vorlage (`jacobi-skeleton.tgz`) des MPI-Programms befindet sich auf der Webseite und ist durch ein Passwort geschützt. Das Passwort erfahren sie in der Vorlesung bzw. in der Übung. Eine Beispielvisualisierungsdatei (`visualization.dat`) ist auch im Archiv enthalten.

Das Jacobi-Verfahren kann gegenwärtig sowohl interaktiv als auch über Kommandozeilenparameter gestartet werden:

```
partdiff [method] [lines] [func] [term] [prec/iter] [ckpt iter] [vis iter]
```

Die genauere semantische Bedeutung der Parameter kann über die Hilfe-Ausgabe des Programms erhalten werden (`./partdiff-par -h`). Wenn Sie die Aufgabe bearbeiten, so ändern Sie **so wenig wie möglich** am Code, und insbesondere ändern Sie **NICHT** die Reihenfolge oder Bedeutung der Kommandozeilenparameter!

Innerhalb des Quellcodes wurde bereits die Routine `writeCheckpoint(void)` angelegt, welche das Checkpointing durchführen soll. Diese wird korrekt angesprochen.

Programmieren Sie die Routine `writeCheckpoint(void)` nach dem traditionellen (MPI-1) I/O-Konzept, in welchem nur ein Master-Prozess die ganze I/O mittels POSIX-Schnittstelle durchführt. Beachten Sie, dass die gesamte Matrix nicht in den Hauptspeicher des Master-Prozesses passen muss, es ist also nicht sinnvoll alle Daten auf einmal an den Master zu transferieren.

Für die Visualisierung wurden die Routinen `initVisualization(void)`, `writeVisualizationData(void)` und `finalizeVisualization(void)` angelegt. Diese Routinen sollen ebenfalls nach dem traditionellen I/O-Konzept mit POSIX-Aufrufen ausprogrammiert werden.

Das Programm muss unabhängig von der Anzahl der Prozesse dieselben Resultate liefern, sonst ist es fehlerhaft!

WICHTIG: Das Programm muss unabhängig von der Anzahl der Prozesse dieselben Resultate liefern!

2.1 Weitere Hinweise:

Definitiv benötigte POSIX-Befehle: `open`, `write`, `close`

Hilfe zu diesen Funktionen erhalten Sie in den `man` pages mit dem Befehl `man <Befehl>` auf den meisten Linux-Systemen.

Es muss sichergestellt werden, dass ein Absturz während dem Checkpointing nicht zur Folge hat, dass die Rechnung von vorne begonnen werden muss. Bedenken Sie aber, dass es aus Speicherplatzgründen nicht sinnvoll ist, zuviele Checkpoints zu speichern.

Versuchen Sie die Visualisierungsdaten mit kollektiven Operationen zum Master zu transferieren.

Testen Sie das Programm auf Korrektheit und verwenden Sie mindestens die verschiedenen Prozesszahlen 1, 2 und 5. Testen Sie auch mindestens 2 verschiedene Größen der Matrix. Protokollieren Sie die dabei gemessenen Laufzeiten - entsprechen diese Ihren Erwartungen?

2.2 Abgabe

Die Abgabe sollte wie folgt strukturiert sein:

- Keine Binär- oder Objekt-Dateien mit abgeben!
- Ein Makefile um das Programm direkt mit `make` übersetzen zu können.
- Der Quelltext soll in einem `.tar.gz`-Archiv (`< nachname_1 >< nachname_2 > ... < nachname_k > .tgz`) gepackt werden.
- Das Archiv soll nur einen Ordner mit dem Quellcode enthalten, der ebenfalls so benannt ist: `< nachname_1 >< nachname_2 > ... < nachname_k >`. Für die Gruppe Hans Mustermann und Klara Musterfrau würde der Ordner also MustermannMusterfrau heißen.
- Hinweis: Das Archiv kann mittels `tar -czf < archiv >< ordner >` erstellt werden.
- Alle Dateien und Ordner sollen keine Sonderzeichen (Leerzeichen, Umlaute etc.) enthalten.
- P.S.: eine umbenannte `.zip`-Datei ist KEINE `.tar.gz`-Datei.

Bei Nichteinhaltung der Abgaberichtlinien wird Ihre Abgabe von Dev Null korrigiert :-)

Bearbeitungszeit			
Schwierigkeit	<input type="radio"/> zu leicht	<input type="radio"/> genau richtig	<input type="radio"/> zu schwer
Lehrreich	<input type="radio"/> wenig	<input type="radio"/> etwas	<input type="radio"/> sehr
Verständlichkeit	<input type="radio"/> großteils unklar	<input type="radio"/> teilweise unklar	<input type="radio"/> verständlich
Kommentar:			

3 Aufgaben (0 Punkte)

Nutzen Sie Ihre Zeit für die Programmierung.

4 Rückmeldung

Gesamte Bearbeitungszeit			
Schwierigkeit	<input type="radio"/> zu leicht	<input type="radio"/> genau richtig	<input type="radio"/> zu schwer
Lehrreich	<input type="radio"/> wenig	<input type="radio"/> etwas	<input type="radio"/> sehr
Verständlichkeit	<input type="radio"/> großteils unklar	<input type="radio"/> teilweise unklar	<input type="radio"/> verständlich
Kommentar:			