Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Sergei Shudler, Sebastian Rinke

Systemnahe und parallele Programmierung (WS 16/17)

Praktikum: MPI

Die Lösungen müssen bis zum 17. Januar 2017 in Moodle submittiert werden. Anschließend müssen sie ihre Lösungen einem Tutor vorführen. Alle Programmieraufgaben müssen auf dem Lichtenberg Cluster kompiliert und ausgeführt werden können. Alle Lösungen müssen zusammen in einer `tar` Datei eingereicht werden.

In diesem Praktikum wird das parallele Sortieren mit MPI behandelt. Das Problem tritt auf, wenn das Eingabearray zu groß ist um in den Speicher eines einzelnen Prozesses zu passen und daher auf alle Prozesse aufgeteilt werden muss. Deshalb muss der Lösungsalgorithmus alle Prozesse einbeziehen und sollte vermeiden die kompletten Eingabedaten in einem Prozess zu sammeln. Formal stellt sich das Problem wie folgt dar:

Eingabe: N ganze Zahlen, die über p Prozesse gleich verteilt sind (es sei $p = 2^k$, $N = 2^l$, $l > k$), so dass Prozess i die folgenden $n = N/p$ Werte enthält: $a_i^{(0)}, a_i^{(1)}, \ldots, a_i^{(n-1)}$.
Ausgabe: Eine verteilte Permutation der Eingabewerte, so dass alle lokalen Werte in jedem Prozess i aufsteigend sortiert sind, und jeder dieser lokalen Werte kleiner oder gleich den Werten in allen Prozessen mit einem höheren Rang j ($j > i$) ist. In anderen Worten gilt also für alle Prozesse $i < j$ und Indizes $q \leq r$: $a_i^{(q)} \leq a_i^{(r)}$ und $a_i^{(n-1)} \leq a_j^{(q)}$.

Allgemeiner Hinweis: Zum Sortieren von lokalen Arrays in MPI Prozessen können Sortierroutinen der C Standard Bibliothek genutzt werden.

Aufgabe 1

(35 Punkte) Angenommen die Eingabe bestehend aus N ganzen Zahlen ist wie oben beschrieben gleichmäßig auf p Prozesse verteilt wobei jeder Prozess n Werte speichert. Implementieren sie ein sehr einfaches paralleles Sortierverfahren indem zuerst das lokale Array in jedem Prozess sortiert wird und anschließend die p Arrays in ein sortiertes Ausgabearray in Prozess 0 vereinigt werden. In diesem Fall wird angenommen, dass alle N Eingabewerte in den Speicher eines Prozesses passen.
Achten sie beim Vereinigen aller Arrays in Prozess 0 darauf, dass dies sowohl hierarchisch als auch parallel durchgeführt werden soll. Wenn zum Beispiel gilt $p = 8$, sollen in der ersten Phase die Arrays der Prozesspaare $\{0,1\}, \{2,3\}, \{4,5\}, \{6,7\}$ vereinigt werden und die Ergebnisse jeweils in den Prozessen $0, 2, 4, 6$ gespeichert werden. In der zweiten Phase werden die Arrays der Paare $\{0,2\}, \{4,6\}$ vereinigt und deren Ergebnisse jeweils in 0 und 4 gespeichert. Zuletzt wird das Paar $\{0,4\}$ vereinigt und das Ergebnis in Prozess 0 gespeichert.

Nutzen sie zur Umsetzung ihrer Lösung die Datei `task1.c` und das Job Skript `batch_job.sh` um einen Job zum Cluster zu submittieren. Überprüfen sie am Ende die korrekte Sortierung der gesamten Daten mit der Funktion `is_arr_sorted`.

Aufgabe 2

(40 Punkte) Das Ziel dieser Aufgabe ist die Verbesserung des Algorithmus aus Aufgabe 1 durch eine auf *Splitter*-Elemente basierende Variante des parallelen Sortierens, wie im Folgenden beschrieben:

1. Jeder Prozess sortiert sein lokales Array.

2. Jeder Prozess wählt $p-1$ Elemente. Der Index von Element i berechnet sich nach $n \cdot (i+1)/p$, $(i = 0, 1, 2, \ldots, p-2)$. Diese Elemente (lokale Splitter) teilen das lokale Array in p gleiche Teile. Der Prozess sendet nun seine $p-1$ lokalen Splitter an den Root Prozess (Prozess 0).

3. Der Root Prozess sammelt die $p \cdot (p-1)$ lokalen Splitter von allen Prozessen, sortiert diese und wählt davon $p-1$ Elemente. Der Index von Element i ist $(p-1) \cdot (i+1)$, $(i = 0, 1, 2, \ldots, p-2)$. Diese Elemente sind nun die globalen Splitter.

4. Der Root Prozess sendet die $p-1$ globalen Splitter an alle anderen Prozesse.

5. Jeder Prozess teilt sein lokales Array den globalen Splittern ensprechend in p Blöcke und sendet anschließend den i-ten Block an Prozess i.

6. Jeder Prozess hat nun p sortierte Arrays die lokal in ein Array vereint werden. Der Einfachheit halber können dafür alle Blöcke in ein Array kopiert werden welches anschließend lokal sortiert wird.

Implementieren sie dieses Verfahren in der Datei `task2.c`. Verwenden sie die kollektiven MPI Operationen `MPI_Gather`, `MPI_Bcast`, `MPI_Alltoall`, `MPI_Alltoallv` und weitere Kollektive sofern sie benötigt werden. Als Beispiel für ein Job Skript können sie `batch_job.sh` nehmen. Verifizieren sie, dass alle Elemente korrekt sortiert sind mit der Funktion `verify_results_eff`.

Implementieren sie eine Verifikationsfunktion `verify_results_eff` die nicht alle Daten auf einem Prozess sammelt. Wenn jeder Prozess sein größtes Element dem nächstgrößeren Rang sendet kann jeder Prozess die Überprüfung zunächst lokal vornehmen. Am Ende der Überprüfung soll ein Prozess ausgeben ob die gesamten Daten korrekt sortiert sind.

Aufgabe 3

(22 Punkte) Diese Aufgabe analysiert die Komplexität des Algorithmus aus Aufgabe 2. Analysieren sie die asymptotische Komplexität im Worst Case (obere Schranke) für die Laufzeit und den Speicherbedarf des Verfahrens unter folgenden Annahmen:

- Jeder Prozess hält genau n Elemente des gesamten Arrays
- Die Anzahl n lokaler Element pro Prozess ist viel größer als die Anzahl der Prozesse p
- Das lokale Sortieren mit der Sortierroutine der C Standard Bibliothek dauert Zeit $O(n \log n)$
- Der Speicherbedarf von MPI Funktionen in der MPI Bibliothek soll nicht berücksichtigt werden
- Die Laufzeitkomplexität einzelner kollektiver MPI Operationen ist wie folgt:
  - `MPI_Bcast` = $O(m \log p)$, m ist die Größe der Daten die der Root Prozess sendet
  - `MPI_Gather` = $O(\log p + m)$, m ist die Summe der Größen der gesammelten Daten im Root Prozess
  - `MPI_Alltoall` = `MPI_Alltoallv` = $O(p + m)$, m ist die Summe der Größen der Daten die ein Prozess zu allen Prozessen sendet

Die Analyse der Laufzeit als auch des Speicherbedarfs sollte nicht zu detailliert sein sondern sich am Detailgrad der gegebenen Beschreibung des Algorithmus aus Aufgabe 2 orientieren. Speichern sie die Lösung dieser Aufgabe in einer PDF Datei mit dem Namen `task3.pdf`.

Aufgabe 4

(3 Punkte) In dieser Aufgabe sollen sie das Tool Extra-P verwenden um ein Modell der Ausführungszeit des Algorithmus aus Aufgabe 2 zu erstellen.
Um Extra-P zu verwenden müssen zuerst mehrere Zeitmessungen für verschiedene Werte des Zielparameters vorliegen. Da wir die Laufzeit als eine Funktion der Eingabegröße betrachten ist unser Zielparameter n, die Anzahl der Elemente pro Prozess. Extra-P benötigt Programmlaufzeiten für mindestens 5 verschiedene Werte von n.

Das bereitgestellte Job Skript (`perf_analysis.sh`) führt das parallele Sortieren für 5 verschiedene Werte von n aus: 2M, 4M, 6M, 8M und 10M. Für jede Eingabe werden 10 Zeitmessungen durchgeführt und die Ausgabe in die Textdatei `input.res` geschrieben (die Eingabedatei für Extra-P). Das Skript verwendet 32 Prozesse (auf 2 Knoten) und nimmt an, dass die ausführbare Datei `task2` heißt und auf der Datei `task2.c` basiert. Entfernen sie bitte nicht die Zeitmessungen aus dem Code, damit das Skript korrekt funktioniert.
Nachdem die Zeitmessungen vom Job Skript erfolgreich beendet sind kann Extra-P auf einem Login Knoten des Lichtenberg Clusters genutzt werden. Dafür führen sie zunächst folgende Befehle auf dem Login Knoten aus:

- `source /home/groups/da_lpp/modules/loadModules.sh`
- `module load extrap/1.0`

Starten sie Extra-P mit `extrap extra-p.conf ./input.res` und untersuchen sie anschließend die erzeugte Ausgabedatei deren Namen auf `.xtrap` endet. Welches Modell erhalten sie für die Laufzeit ihres Algorithmus? Wie unterscheidet es sich von ihrer analysierten asymptotischen Laufzeit aus Aufgabe 3? Speichern sie die Lösung dieser Aufgabe in einer PDF Datei mit dem Namen `task4.pdf`.