Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Sergei Shudler, Sebastian Rinke

Systemnahe und Parallele Programmierung (WS 18/19)

Praktikum: MPI

Die Lösungen müssen bis zum 15. Januar 2019, 13:30 Uhr, in Moodle submittiert werden. Anschließend müssen Sie ihre Lösung einem Tutor vorführen. Das Praktikum wird benotet. Im Folgenden einige allgemeine Bemerkungen, die für alle Aufgaben auf diesem Blatt gelten:

- Es gibt ein vorgegebenes Makefile welches alle Programme kompiliert die Sie entwickeln sollen.
- Dynamisch allokierter Speicher muss freigegeben werden bevor das Programm endet.
- Die Programme müssen auf dem Lichtenberg Cluster kompilierbar und ausführbar sein. Alle Zeitmessungen müssen auf dem Lichtenberg Cluster ausgeführt werden. Es kann ein Beispiel Job Skript von Moodle heruntergeladen werden. Es enthält auch Hinweise, wie Sie es für Ihre Bedürfnisse anpassen können.
- Falls in der Aufgabenstellung nicht explizit erlaubt, darf vorgegebener C Code in den Dateivorlagen nicht geändert und muss wie gegeben verwendet werden.
- Alle Lösungen die keinen Programmcode erfordern, bitte zusammen in einer PDF Datei einreichen.
- Laden Sie alle Dateien in einem tar Archiv in Moodle hoch.

Dieses Praktikum befasst sich mit dem parallelen Sortieren von verteilten Daten. Wir nehmen an, dass n Elemente über p Prozesse verteilt sind, wobei $n < p$. Demnach besitzen nicht alle Prozesse Daten und viele nur eines der n zu sortierenden Elemente. Weiterhin gibt es keine zwei Elemente mit dem gleichen Wert. Da die Anzahl der Prozesse p sehr groß sein kann, ist es nicht praktikabel alle Datenelemente in einem Prozess zu sammeln, um sie dann sequenziell zu sortieren. Im Folgenden wird ein paralleler Algorithmus für das Problem beschrieben.

Der Algorithmus nimmt an, dass die Anzahl der Prozesse p eine Quadratzahl ist, d.h., $p = d^2$, $d \in \mathbb{N}$. Alle Prozesse sind in einem zweidimensionalen $d \times d$ Gitter angeordnet. Zunächst sammelt jeder Prozess alle Elemente von Prozessen die sich in der gleichen Zeile des Gitters wie er befinden. Dasselbe macht er für seine Spalte im Gitter. Alle Elemente der gleichen Zeile nennen wir *Zeilenelemente* und alle Elemente der gleichen Spalte entsprechend *Spaltenelemente*.

Seine Zeilen- und Spaltenelemente speichert jeder Prozess in zwei separaten Feldern. In der Abbildung unten werden diese Felder durch [row]/[col] dargestellt, wobei [row] die Zeilenelemente und [col] die Spaltenelemente enthalten. Die Felder [row] und [col] werden separat aufsteigend sortiert.

Im nächsten Schritt ermittelt jeder Prozess den lokalen Rang (rank) von jedem Spaltenelement in [col], entsprechend der Position im Feld [row]. Der Rang ist der Index, an dem das Spaltenelement im Feld [row] eingefügt werden müsste, ohne die aufsteigende Sortierung von [row] zu verletzen. Nach dem Berechnen der lokalen Ränge der Spaltenelemente, wird der globale Rang von jedem Spaltenelement ermittelt. Dies wird erreicht, indem die lokalen Ränge entlang jeder Spalte aufsummiert werden. Der globale Rang entspricht der Position in einem gedachten globalen Feld, das alle n Elemente aufsteigend sortiert enthält.

Schließlich werden die Element zwischen den Prozessen so umverteilt, dass Prozess r das Element mit globalem Rang r speichert. Die Reihenfolge der Elemente von Prozess 0 bis $p-1$ ist nun aufsteigend sortiert.

$$\begin{pmatrix} [c] & [] & [] & [h] \\ [] & [a] & [e] & [] \\ [] & [g] & [] & [b,d] \\ [i] & [f,j] & [] & [] \end{pmatrix}$$

$\downarrow$ Allgather, [row]/[col]

$$\begin{pmatrix} [c,h]\,/\overset{0\ 2}{[c,i]} & [c,h]\,/\overset{0\ 1\ 1\ 2}{[a,f,g,j]} & [c,h]\,/\overset{1}{[e]} & [c,h]\,/\overset{0\ 1\ 1}{[b,d,h]} \\ [a,e]\,/\overset{1\ 2}{[c,i]} & [a,e]\,/\overset{0\ 2\ 2\ 2}{[a,f,g,j]} & [a,e]\,/\overset{1}{[e]} & [a,e]\,/\overset{1\ 1\ 2}{[b,d,h]} \\ [b,d,g]\,/\overset{1\ 3}{[c,i]} & [b,d,g]\,/\overset{0\ 2\ 2\ 3}{[a,f,g,j]} & [b,d,g]\,/\overset{2}{[e]} & [b,d,g]\,/\overset{0\ 1\ 3}{[b,d,h]} \\ [f,i,j]\,/\overset{0\ 1}{[c,i]} & [f,i,j]\,/\overset{0\ 0\ 1\ 2}{[a,f,g,j]} & [f,i,j]\,/\overset{0}{[e]} & [f,i,j]\,/\overset{0\ 0\ 1}{[b,d,h]} \end{pmatrix} \downarrow \text{Sum ranks}$$

$r[c]=2$ $r[i]=8$	$r[a]=0$ $r[f]=5$ $r[g]=6$ $r[j]=9$	$r[e]=4$	$r[b]=1$ $r[d]=3$ $r[h]=7$

Aufgabe 1

(30 Punkte) Implementieren Sie den oben beschriebenen Algorithmus für Integer Elemente mit MPI in der vorgegebenen Datei `task1.c`. Stellen Sie sicher, dass kein Prozess alle Elemente oder all lokalen Ränge von einem Element speichert. Das heißt, der Speicherverbrauch von jedem Prozess ist $O(d)$ und nicht $O(n)$.

Die folgenden und weitere MPI Funktionen werden Sie benötigen:

- `MPI_Comm_split` – partitioniert die Prozesse eines Communicators in disjunkte Gruppen. Jede Gruppe enthält alle Prozesse, die diese Funktion mit dem gleichen Farbargument aufrufen. Für jede Gruppe wird ein neuer Communicator erstellt. Mehr Informationen finden Sie auf:
  `http://mpi.deino.net/mpi_functions/MPI_Comm_split.html`
- `MPI_Allgather` – Mehr Informationen finden Sie auf:
  `http://mpi.deino.net/mpi_functions/MPI_Allgather.html`
- `MPI_Allgatherv` – Mehr Informationen finden Sie auf:
  `http://mpi.deino.net/mpi_functions/MPI_Allgatherv.html`

Ihre Implementierung umfasst folgende Teilaufgaben:

1. **(3 Punkte)** Erstellen Sie Communicators für die Kommunikation entlang der Zeilen und der Spalten. Nutzen Sie die Funktion `MPI_Comm_split`.
2. **(6 Punkte)** Nutzen Sie `MPI_Allgather` und `MPI_Allgatherv` um die Zeilen- und Spaltenelemente zu sammeln.
3. **(2 Punkte)** Sortieren Sie die Zeilen- und Spaltenelemente mit der Funktion `qsort()` aus der C-Standardbibliothek.

4. **(4 Punkte)** Berechnen Sie die lokalen Ränge der Spaltenelemente. Stellen Sie sicher, dass Ihre Implementierung nur Zeit $O(a+b)$ benötigt, wobei a die Anzahl Elemente im Feld [row] und b die Anzahl im Feld [col] sind.

5. **(4 Punkte)** Bestimmen Sie die globalen Ränge der Spaltenelemente in [col] mit `MPI_Allreduce`.

6. **(6 Punkte)** Verteilen Sie die Datenelemente, die jeder Prozess als Eingabe erhalten hat, so um, dass der Prozess mit Rang r das Element mit dem globalen Rang r empfängt. Hierbei ist dem Empfänger der Rang des Senders nicht bekannt. Die vorgegebene Datei nutzt die MPI Routinen `MPI_Isend` und `MPI_Waitall` für das Senden der Elemente. Diese Routinen sind hier notwendig um einen Deadlock im Programm zu vermeiden.

   Genauer gesagt blockiert `MPI_Isend` nicht bis der Empfänger die Nachricht erhalten hat, sondern kehrt immer sofort nach dem Aufruf zurück. Jede mit `MPI_Isend` versandte Nachricht erzeugt einen Request. Die Funktion `MPI_Waitall` erhält ein Feld von diesen Requests und kehrt erst zurück wenn die entsprechenden Nachrichten versandt wurden. Mehr Informationen zu `MPI_Isend` und `MPI_Waitall` finden Sie auf `http://mpi.deino.net/mpi_functions/`.

   Implementieren Sie das Senden der Elemente unter Nutzung von `MPI_Isend` und `MPI_Waitall`. Die vorgegebenen Codefragmente sollen Ihnen die Implementierung erleichtern und müssen an den Rest Ihres Programms angepasst werden. Das heißt, Codeanpassungen sind hier erlaubt. Schließlich muss noch der Code für den Empfänger entwickelt werden.

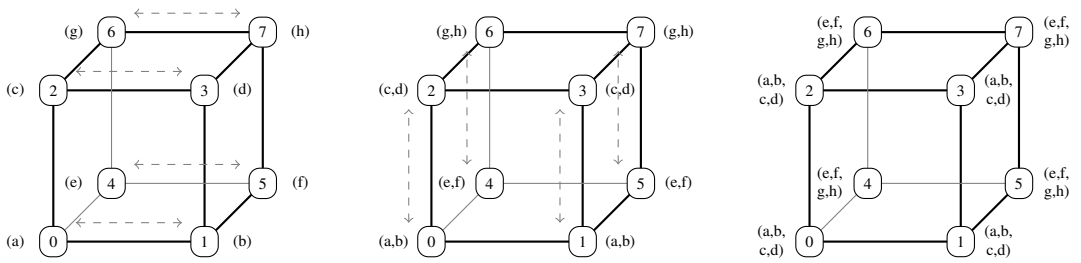
7. **(5 Punkte)** Implementieren Sie die Funktion `verify_results` und nutzen Sie diese um die Korrektheit der Sortierung zu überprüfen. Stellen Sie sicher, dass kein Prozess alle Elemente sammelt. Nutzen Sie unter anderem `MPI_Sendrecv` mit `MPI_PROC_NULL`.

In der Datei `task1.c` ist der Code zum Initialisieren der Eingabe bereits vorgegeben. Jeder Prozess erhält höchstens drei Elemente, wobei die Gesamtzahl der Elemente kleiner als die Anzahl Prozesse ist. Nutzen Sie das Batch Skript `batch_job.sh` um Ihr Programm auf dem Cluster auszuführen.

Aufgabe 2

(20 Punkte) In dieser Aufgabe modifizieren wir die Lösung von Aufgabe 1, indem wir das Sammeln der Zeilen- bzw. Spaltenelemente mit dem Sortieren dieser Elemente kombinieren. Das heißt, die Teilaufgaben 2 und 3 von Aufgabe 1 werden vereinigt. Dafür implementieren Sie einen *Allgather-Merge* Algorithmus mit MPI Point-to-Point Kommunikation. Die Voraussetzung für den Algorithmus ist, dass für die Anzahl Prozesse p gilt, $p = 2^k$. Die Prozesse sind in einem k-dimensionalen Hypercube angeordnet. Somit hat jeder Prozess k Nachbarn, einer in jeder Dimension.

Die folgende Abbildung zeigt ein Beispiel für 8 Prozesse. In Klammern stehen die Datenelemente von jedem Prozess. In der ersten Iteration sendet jeder Prozess alle seine Elemente an den Nachbarn entlang der ersten Dimension des Hypercubes und empfängt alle Elemente vom selben Nachbarn. Die empfangenen Elemente werden dann mit den bereits vorhandenen Elementen vereint (merge), so dass die Elemente in jedem Prozess wieder sortiert sind. In der zweiten Iteration werden die Elemente mit dem Nachbarn in der zweiten Dimension ausgetauscht und wieder vereint, und so weiter. Bei k Dimensionen gibt es k Iterationen. Nach der k-ten Iteration besitzt schließlich jeder Prozess die Elemente von allen Prozessen in aufsteigend sortierter Reihenfolge. Der Algorithmus führt $O(\log p)$ Kommunikationsschritte pro Prozess aus. Unser Beispiel zeigt nur zwei Iterationen. Nach der dritten (letzten) Iteration besitzt jeder Prozess alle Elemente in der Ordnung (a,b,c,...,h).



Beachten Sie, dass wir die Annahme $p = d^2$ aus Aufgabe 1 noch berücksichtigen müssen. Das heißt, kombiniert mit der Annahme $p = 2^k$ von unserem Allgather-Merge Algorithmus, muss $p = 4^l$, $l \in \mathbb{N}$, sein. Nutzen Sie für Ihre Implementierung die Datei `task2.c` und das Beispiel Job Skript `batch_job.sh`. Die Implementierung besteht aus folgenden Teilaufgaben:

1. **(2 Punkte)** Sortieren Sie die lokalen Eingabeelemente von jedem Prozess. Dies ist notwendig für den Allgather-Merge Algorithmus, da jeweils die vorhandenen und empfangenen Elemente als sortiert angenommen werden.

2. **(4 Punkte)** In der Funktion `all_gather_merge`: Berechnen Sie den MPI Rang des Kommunikationspartners in jeder Iteration. Der Rang in Iteration i ist die binäre XOR Verknüpfung des eigenen Rangs mit 2^i, wobei $0 \leq i \leq k-1$ und k der Anzahl Dimensionen des Hypercubes entspricht.

3. **(6 Punkte)** In der Funktion `all_gather_merge`: Jeder Prozess sendet seine aktuellen Elemente an den Kommunikationspartner und empfängt dessen Elemente. Nutzen sie dazu `MPI_Sendrecv`.

4. **(4 Punkte)** In der Funktion `all_gather_merge`: Jeder Prozess vereint (merge) die vorhandenen Elemente mit den in der Iteration Empfangenen, so dass in der nächsten Iteration die vereinten aktuellen Elemente zum nächsten Nachbarn gesendet werden können. Nutzen Sie die vorgegebene Funktion `merge_arr`.

5. **(4 Punkte)** Rufen Sie die Funktion `all_gather_merge` jeweils auf um die Zeilen- und Spaltenelemente aufsteigend sortiert zu erhalten.

Aufgabe 3

(5 Punkte) In dieser Aufgabe werden Sie das Tool Extra-P verwenden, um ein Leistungsmodell für die Laufzeit der Lösung von Aufgabe 1 zu bestimmen.

Um Extra-P zu nutzen, müssen Sie zuerst mehrere Messungen, für unterschiedliche Werte, mit Ihrem Zielparameter durchführen. Da wir ein Modell für die Laufzeit in Abhängigkeit von der Anzahl Prozesse berechnen möchten, ist unser Zielparameter die Anzahl Prozesse p. Extra-P benötigt Laufzeitmessungen für mindestens fünf unterschiedliche Werte von p. Das Batch Skript `perf_analysis.sh` misst die Zeit von Ihrem Programm für die folgenden Werte von p: 4, 9, 16, 25 und 36. Für jedes p führt das Skript zehn Iterationen aus und schreibt die Ergebnisse in die Datei `input.res`. Diese Datei ist eine Textdatei und die Eingabe für Extra-P. Ihr ausführbares Programm muss `task1` benannt sein. Damit das Skript korrekt funktioniert, entfernen Sie bitte nicht die Zeitfunktionen in Ihrem Code. Um nach den Laufzeitmessungen eine Modell mit Extra-P zu berechnen, führen Sie folgende Schritte auf einem Login Knoten aus:

1. `source /home/kurse/kurs00015/da_lpp/modules/loadModules.sh`
2. `module purge`
3. `module load gcc/4.9.4 intel/2016u4 papi/5.5.1 openmpi/intel`
4. `module load scorep/intel/ompi/1.4.2 python`
5. `module load extrap/2.0`

Wir nutzen hier Extra-P auf der Befehlszeile. Eine grafische Oberfläche existiert jedoch auch. Das Modell wird nun durch den Aufruf `extrap_cmd input.res` generiert. Die Ausgabe besteht aus sechs Feldern: <metric name>, <region name>, <model>, <no. terms>, <adj. R-squared>, <SMAPE>. Unser Modell ist im dritten Feld. Welches Modell erhalten Sie?