Institut für Numerische und Angewandte Mathematik 21.12.2017
FB Mathematik und Informatik der Universität Münster
Prof. Dr. Christian Engwer, MSc. Marcel Koch

Übung zur Vorlesung

Wissenschaftliches Rechnen

WS 2017/18 — Blatt 11

Abgabe: 11.01.2018, 10:00 Uhr, Briefkasten 111
Code zusätzlich per e-mail an `marcel.koch@uni-muenster.de`

Auf der Vorlesungsseite haben wir Ihnen Hinweise zur Verwendung von `MPI` im Computerpool SRA zusammengestellt. Lesen Sie diese vor der Bearbeitung der nächsten Aufgabe und führen Sie den Punkt *System einrichten* durch. Wie im Punkt *Erste Schritte mit MPI* beschrieben, können Sie anschließend testen, ob alles ordnungsgemäß funktioniert.

Aufgabe 1 (Bonusaufgabe: Parallelisierung des Turingmodells mit MPI) (8 Punkte)

Betrachten Sie die Implementierung von Aufgabe 3 auf Blatt 8 zusammen mit dem konkreten Modell zur Belousov-Zhabotinsky Reaktion aus Aufgabenteil (c). Parallelisieren Sie den Code, indem Sie das Rechengebiet Ω in Teilgebiete zerlegen, welche parallel bearbeitet werden. Die Teilgebiete sollen möglichst „rund“ sein, d.h. die Kantenlängen jedes Teilgebiets sollen möglichst gleich groß sein.

- Die in einem konkreten Teilgebiet liegenden Gitterzellen sollen von einem einzelnen Prozess bearbeitet werden. Aus Sicht dieses Prozesses wollen wir sie deshalb *owner-Zellen* betiteln. Die Funktion für die rechte Seite jeder Komponente des mit dem zellzentrierten Finite-Volumen-Verfahren semidiskretisierten Diffusionsproblems benötigt alle Nachbarn der owner-Zellen. Jedes Teilgebiet sollte also in Richtung der angrenzenden Teilgebiete jeweils um eine Reihe von Gitterzellen erweitert werden, welche eigentlich in einem anderen Teilgebiet liegen, d.h. owner-Zellen eines anderen Prozesses sind. Derartige Gitterzellen nennen wir *overlap-Zellen*. Die lokalen Daten eines Prozesses sollten also sowohl in owner-Zellen als auch in overlap-Zellen liegen, die Berechnungen hingegen auf owner-Zellen eingeschränkt werden.

- Nach der Berechnung eines neuen Zeitschrittes steht für overlap-Zellen der falsche Wert im lokalen Lösungsvektor. Um den richtigen Wert zu erhalten, müssen die Daten in overlap-Zellen mit den Nachbarprozessen ausgetauscht werden. Bei der Kommunikation sollten so wenig Daten wie möglich verschickt werden, d.h. wirklich nur die Daten in overlap-Zellen.

- Die Klasse zur Generierung von Gitterinformationen für Finite-Volumen-Verfahren auf kartesischen Gittern sollte derart erweitert werden, dass sie zusätzlich die für die Parallelisierung nötigen Informationen über die parallele Gitteraufteilung liefert. Die

- Um die Kommunikation zwischen den Prozessen zu vereinfachen, ordnen Sie die Prozesse ebenfalls in einem kartesischem Gitter an. Dazu müssen Sie den Rang eines Prozesses eindeutig einen 2 dimensionalen Index zu ordnen. Sie können selbstgeschriebene Funktionen

dafür nutzen, oder Sie greifen auf `MPI` Funktionen zurück. Zur Behandlung von Prozessen in einem kartesischem Gitter stellt `MPI` folgende Funktionen zur verfügung:

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

liefert eine mögliche Aufteilung von `nnodes` Prozessen auf ein `ndims` dimensionales Gitter, welche in `dims` zurück gegeben wird. So werden zum Beispiel 6 Prozesse auf einem $2D$ Gitter in 3×2 partitioniert. Beachten Sie, dass `dims` mit Null initialisiert sein muss, da nur Einträge in `dims` gleich Null verändert werden.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

erzeugt einen neuen `MPI` Kommunikator `comm_card` mit Prozessen aus dem alten Kommunikator `comm_old`, der die Prozesse auf ein kartesisches `ndims` dimensionales Gitter abbildet. Die Anordnung der Prozesse auf dem Gitter ist durch `dims` gegeben. In $2D$ entspricht dabei `dims[0]` dem y Index und `dims[1]` dem x Index. In `periods` wird festgelegt, ob die Ränder des Gitters periodisch sind, d.h. ob Daten, die z.B. am oberen Rand bildlich gesprochen nach oben gesendet werden, am unteren Rand ankommen. Sie sollten dieses Array auf Null setzten, um die Periodizität zu deaktivieren. Der Parameter `reorder` gibt an, ob die Ränge der Prozesse geändert werden darf oder nicht, setzen Sie diesen Wert einfach auf Null.

```
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

erlaubt die Umwandlung zwischen dem Rang eines Prozesses und dessen Koordinaten im kartesichen Gitter. Es muss gelten $coords[i] < dims[i]$. Damit lässt sich leicht feststellen, mit welchen Prozessen kommunitiert werden muss. Möchte man zum Beispiel mit dem Prozess rechts von einem, d.h. in `dims[0]` Richtung $+1$, kann man folgendes Konstrukt verwenden

```
    int my_coords[2] = {0,0};
    MPI_Cart_coords(comm, my_rank, my_coords);

    int right_rank;
    int right_coords[2] = {my_coords[0],my_coords[1]+1};
    MPI_Cart_rank(comm, right_coords, right_rank);
```

Die folgende Funktion ermöglicht es beide Nachbarn in einer Richtung eines Prozesses gleichzeitig zu bestimmen.

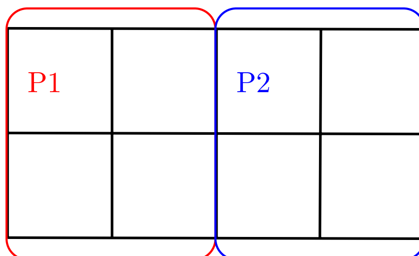
```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest)
```

In `rank_source` befindet sich der Rang des Prozess mit den Koordinaten

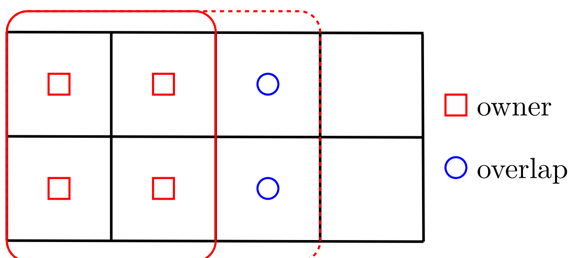
```
coords = my_coords; coords[direction] -= disp;
```

und in `rank_dest` entsprechend der Rang des Prozesse mit Koordinaten `coords[direction] += disp`. Falls `direction=1` und `disp=1` entsprich `rank_source` dem Rang des Prozesses links von einem und `rank_dest` dem Rang des Prozesses rechts von einem. Befindet sich der Prozess, der diese Funktion aufruft, auf dem Rand des Gitters bezüglich der `direction` Richtung dann enthält einer der Rückgabewerte `MPI_PROC_NULL` um zu zeigen, dass dort kein Nachbar ist. Bei dem Datenaustausch kann dieses Makro benutzt werden sowie ein gültiger Rang, nur das in diesem Fall ein Aufruf einer Send- oder Recive-Funktion keinen Effekt hat.

- Das Schreiben der Visualisierungdaten sollte von einem einzelnen Prozess durchgeführt werden, um Koordinationsprobleme beim Zugriff auf die Datei zu vermeiden. Die einfachste Möglichkeit ist es also, an dieser Stelle des Codes auf eine Parallelisierung zu verzichten und alle Prozesse eine Kopie ihres kompletten lokalen Lösungsvektors an einen ausgezeichneten Prozess schicken zu lassen, welcher dann alle Daten in eine Datei schreibt.



Parallele Aufteilung der Gitterzellen auf zwei Prozesse P1, P2.



Vergrößerung des Teilgebietes von P1 sowie Identifizierung der owner- und overlap-Zellen.

Hinweis 1: Ohne eine Parallelisierung des Datenoutputs sollte man sich auf das Herausschreiben der Lösung zum finalen Zeitpunkt beschränken.

Hinweis 2: Soll die empfangene Kopie des kompletten lokalen Lösungsvektors eines Prozesses direkt nach dem Empfang in die Datei geschrieben werden, d.h. bevor die Daten sämtlicher Prozesse empfangen wurden, ist zu beachten dass die Daten nicht einfach hinten an die Datei angehängt werden dürfen. In diesem Fall muss an die richtige Stelle innerhalb der Datei gesprungen werden, wofür der Befehl `fseek` benutzt werden kann.