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

02.12.2019

Übung zur Vorlesung

Wissenschaftliches Rechnen

WS 2019/20 — Blatt 8

Abgabe: 09.12.2019, 10:00 Uhr Aufgabe 1-3, Briefkasten 112
06.01.2020. 10:00 Uhr Aufgabe 4
Code zusätzlich per e-mail an `marcel.koch@uni-muenster.de`

Achtung: Achten Sie darauf, Ihre Programme ordentlich zu formatieren und gut zu kommentieren. Die Form wird mit in die Bewertung eingehen.

Aufgabe 1 (Multikative Schwarz-Methode) (4 Punkte)

Sei I die Indexmenge aller innerer Knoten und $I_i := \{i\}$ eine nicht-überlappende Zerlegung von I, so dass jede Indexmenge genau einen Knoten enthält. Zeigen Sie, dass die multiplikative Schwarz-Methode angewandt auf diese Zerlegung äquivalent ist zu der Gauß-Seidel Iteration

$$x^{k+1} = x^k + W^{-1}(b - Ax^k),$$

mit der additiven Zerlegung von $A = L + D + U$ und $W = D + L$.
Verallgemeinern Sie dieses Ergebnis auf den Fall $I_i \cap I_j = \emptyset \ \forall i \neq j$, um zu zeigen, dass in dem Fall die multiplikative Schwarz-Methode äquivalent zu einer Block-Gauß-Seidel Iteration ist.

Aufgabe 2 (Additive Schwarz-Methode) (4 Punkte)

Bei der additiven Schwarz-Methode ist die Iterationsvorschrift gegeben durch

$$x^{k+1} = x^k + \omega \sum_{i=1}^{p} R_i^T A_i^{-1} R_i (b - Ax^k), \quad \omega \in \mathbb{R},$$

wobei R_i, A_i wie bei der multiplikativen Variante definiert sind. Zeigen Sie, dass die Matrix

$$M^{-1} = \sum_{i=1}^{p} R_i^T A_i^{-1} R_i, \text{ mit } A_i = R_i A R_i^T,$$

die sich als Vorkonditionierer der additiven Schwarz-Methode für eine gegebene Matrix A auffassen lässt, positiv definit ist, falls A positiv definit ist.

Die folgenden Programmieraufgaben befassen sich mit MPI. Das Codegerüst für diese Aufgaben beinhält auch ein Hello-World Programm für MPI. Sie können zur weiteren Dokumentation von MPI Befehlen unteranderem Cheat-Sheets[1] oder die offizielle Dokumentation[2] hinzu nehmen. Um ein MPI Programm zu kompilieren muss ein spezieller Kompiler verwendet werden. In dem Codegerüst wird bereits der korrekte Kompiler ausgewählt, Sie müssen sich also darüber nicht weiter kümmern. Zur Ausführung eines MPI Programms muss ein spezielles Programm verwendet werden, `mpiexec [-n P] executable`. Auf den Rechnern im SR-A ist dieses Programm (und der passende Kompiler) bereits installiert.

Aufgabe 3 (MPI Kommunikations-Varianten) (4 Punkte)

Schreiben Sie ein MPI Programm, das folgende Schritte durchführt:

(a) Jeder Prozess generiert sich eine Integer-Zahl zwischen $0, \cdots, 99$. Diese wird dann an einen vorher bestimmten Prozess gesendet, z.B. der Prozess mit Rang 0. Dieser Prozess gibt dann die Zahlen aus, sortiert nach dem Prozess der sie gesendet hat. (Dies benötigt eine "alle-an-einen" Kommunikation.)

(b) Der Prozess mit Rang i berechnet den absoluten Betrag der Differenz zwischen seiner Zahl und der Zahl vom Prozess $i - 1$. Der erste Prozess vergleicht dabei seine Zahl mit der des letzten Prozesses. (Dies benötigt eine "punkt-zu-punkt" Kommunikation im Ring).

(c) Abschließend soll die maximale Differenz von Zahlen zweier benachbarter Prozesse ermittelt werden. Das Ergebniss soll wieder an alle Prozesse gesendet werden und der Prozess, an dem die maximale Differenz erreicht wurde soll eine Nachricht ausgeben. (Dies benötigt neben einer "alle-an-einen" zusätzlich eine "eine-an-alle" Kommunikation.)

Für diese Aufgabe werden Sie unteranderem folgende Befehle benötigen:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dst, int tag,
             MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag,
             MPI_Comm comm, MPI_Status *status);
```

Die zu empfangenen oder sendenen Daten werden in `buf` übergeben, wobei `count` angiebt wie viele Daten übertragen werden sollen. Für diese Aufgabe wird `count` immer 1 sein. `datatype` gibt an, welchen Typen die Daten haben, hier sollte dies `MPI_INT` sein. Der Rang des Ziel- oder Quell-Prozesses wird an `dst`, oder `src` übergeben, wobei damit der Rang bzgl. der durch `comm` definierten Prozessor-Gruppe gemeint ist. Hier kann einfach die Standard-Gruppe `MPI_COMM_WORLD` verwendet werden. Der Tag zweier zusammemgehöriger Kommunikationen (send und recv) muss bei beiden Aufrufen der gleiche sein.

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                MPI_Op op, int root, MPI_Comm comm);
```

Damit können Daten von allen Prozessen an einen Prozess (`root`) gesendet und gleichzeitig verarbeitet werden. Wie die Daten verarbeitet werden, ist in `op` definiert, z.B. kann durch

[1] http://kayaogz.github.io/teaching/app4-programmation-parallele-2018/cours/MPI-cheatsheet.pdf

[2] http://www.mpich.org/static/docs/latest/

`MPI_SUM` die Summe über alle Werte gebildet werden oder über `MPI_MAX` das Maximum über alle Werte. Die zu sendenen Daten werden an `sendbuf` übergeben, die empfangenen Daten werden in `recvbuf` gespeichert. Dabei ist zu beachten, dass nach dieser Funtkion nur in dem Prozess mit Rang `root` `recvbuf` Daten enthält, in den anderen Prozessen sollte nicht davon ausgegangen werden, dass `recvbuf` sinnvolle Daten enthält.

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

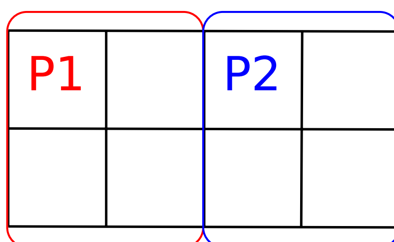
Diese Funktion sendet die Daten von Rang `root` in `buf` an alle Prozesse.

Hinweis: Die Daten zur Übertragung werden durch Pointer übergeben. Bei einer lokalen Variable `int a`*, erhält man einen Pointer, der auf diese Variable zeigt durch* `&a`*. Bei* `std::vector` *oder* `std::array` *Objekten, kann man den Pointer auf die Daten durch die Klassenfunktion* `data()` *erhalten.*

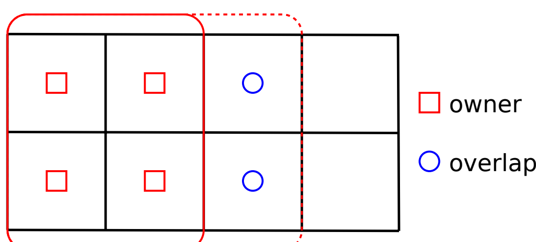
Aufgabe 4 (Parallelisierung des Turingmodells mit MPI) (12 Punkte)

Betrachten Sie die Implementierung von Aufgabe 3 auf Blatt 5 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.



Parallele Aufteilung der Gitterzellen auf zwei Prozesse P1, P2.



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

- 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.