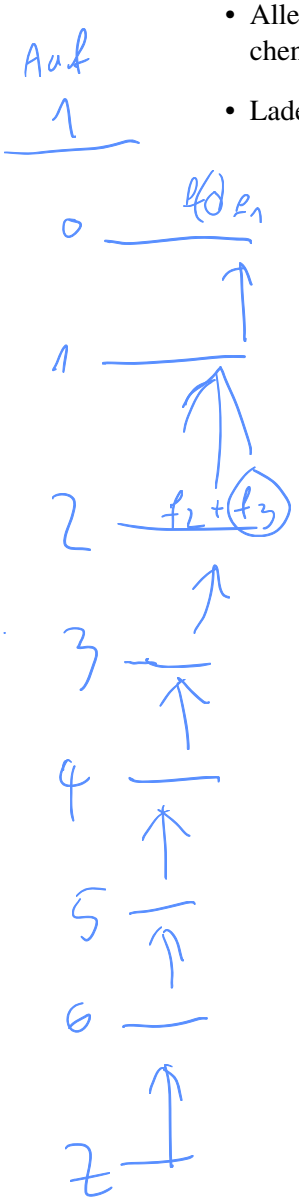
Alexander Hück et al., spp-support@lists.sc.informatik.tu-darmstadt.de

Systemnahe und parallele Programmierung (WS 22/23)

Exercise MPI

Die Lösungen müssen bis zum 17. Januar 2023, 16:00 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 jeweils ein vorgegebenes Makefile welches das Programm kompiliert, das 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. Siehe Anleitung zur Nutzung des Clusters.
- Alle Lösungen, die keinen Programmcode erfordern, bitte zusammen in einer PDF Datei einreichen.
- Laden Sie alle Dateien in einem zip-Archiv in Moodle hoch.



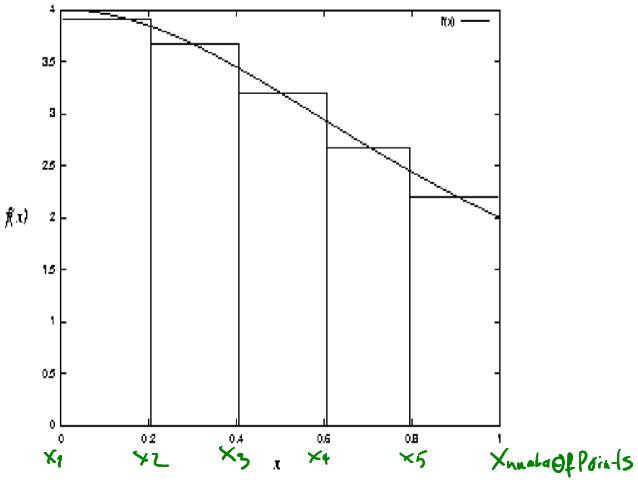
Erläuterung der nicht-blockierenden MPI Routinen

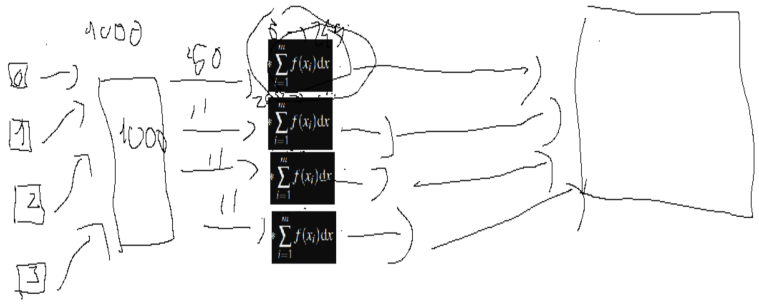
In der Vorlesung wurden blockierende Kommunikationsroutinen behandelt, wie z. B. `MPI_Send` und `MPI_Recv`. Der aufrufende Prozess blockiert die Ausführung dabei so lange, bis der Buffer von der MPI Bibliothek ausgelesen wurden (Senderseite) bzw. die Nachricht empfangen wurde (Empfängerseite).
Dies garantiert zwar einerseits, dass die Daten bei der anschließenden Verwendung vollständig vorliegen. Andererseits kann der Prozess, während er auf den Abschluss der Kommunikation wartet, keine weitere Arbeit verrichten.
In vielen Anwendungsfällen ist es aber grundsätzlich sinnvoll, während dieser Wartezeit weitere Berechnungen durchzuführen, die nicht von den ausgetauschten Daten abhängen. In diesem Fall ist die Verwendung von blockierender Kommunikation ineffizient und kann die Performance negativ beeinflussen. Dieses Problem kann durch die Verwendung von nicht-blockierenden (auch asynchron genannten) MPI Routinen gelöst werden. Anders als bei den blockierenden Varianten, wartet der Prozess nicht auf den Abschluss der Kommunikation, sondern fährt sofort mit der Ausführung des Programms fort. Es ist dann die Aufgabe des Programmierers, sich zu überlegen, an welchen Stellen eine Synchronisierung nötig ist.

Die wichtigsten nicht-blockierenden Operationen sind `MPI_Isend` und `MPI_Irecv`. Die Verwendung ist dabei sehr ähnlich zu den blockierenden Varianten. Die Funktionssignatur unterscheidet sich lediglich dadurch, dass ein zusätzlicher Pointer zu einem `MPI_Request` Objekt mitgegeben wird, welches zur Identifikation der Kommunikation dient. Zum Abschluss der Kommunikation wird das Request Objekt an die `MPI_Wait` Funktion gegeben, welche die Ausführung so lange blockiert, bis alle Daten vollständig ausgetauscht wurden. Es ist wichtig, dass `MPI_Wait` aufgerufen wird, bevor der betreffende Buffer ausgelesen oder modifiziert wird, da sonst noch alte oder unvollständige Werte vorliegen können (Race Condition).

Wichtige Funktionen

Funktionssignatur	Erklärung
`int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)`	Verwendung wie `MPI_Send`. Erwartet zusätzlich ein `MPI_Request` Objekt.
`int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)`	Verwendung wie `MPI_Recv`. Erwartet zusätzlich ein `MPI_Request` Objekt.
`int MPI_Wait(MPI_Request *request, MPI_Status *status)`	Wartet auf den Abschluss der Kommunikation, die zu dem übergebenen `MPI_Request` Objekt gehört. Der Status Parameter ist nur für die Fehlerbehandlung nötig und kann auf `MPI_STATUS_IGNORE` gesetzt werden.





Task 1

Approximation eines Integrals (18 Punkte) In dieser Aufgabe soll eine Monte-Carlo-Simulation zur Berechnung von Integralen implementiert werden.

Wird betrachten das Flächenintegral einer Funktion f über das Intervall $[a,b] \in \mathbb{R}$. Es werden nun m zufällig verteilte Punkte $x_i \in [a,b]$ gewählt, an denen die Funktion ausgewertet wird. Der Mittelwert aller Funktionswerte $f(x_i)$ kann dann verwendet werden, um das Integral zu approximieren. Für große Werte von m gilt dann:

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{m} * \sum_{i=1}^{m} f(x_i)\mathrm{d}x$$

Implementieren Sie eine mit MPI parallelisierte Version der beschriebenen Monte-Carlo-Simulation. Nutzen Sie dann Ihren Code, um das folgende Integral zu berechnen:

$$\int_{-1}^{1} \frac{2}{1+x^2}$$

Als Grundlage dient dabei die Datei `integral.c`, die mit dem beigefügten Makefile kompiliert werden kann. Verwenden Sie in Ihrer Lösung **NICHT** die Befehle `MPI_Reduce` oder `MPI_Allreduce`[1]. Implementieren Sie diese Funktionalität stattdessen selbst in einer eigenen Reducefunktion mit Hilfe von Send/Receive Operationen. Achten Sie dabei darauf, dass das Zusammenführen der Ergebnisse der verschiedenen Prozesse möglichst effizient (im Hinblick auf die Anzahl der nötigen Kommunikations- und Berechnungsoperationen) abläuft. Bei n Prozessen sollte jeder Prozess max. $log_2(n)$ Kommunikationen für die Reduktion benötigen (ggf. einige wenige mehr, falls n keine Zweierpotenz). Zusätzlich ist erneut darauf zu achten, dass jeder Prozess einen anderen random seed verwendet.
Ihre Lösung soll mit einer beliebigen Anzahl an Prozessen ausführbar sein, solange die Anzahl der Prozesse $\leq$ der Anzahl der betrachteten Punkte ist.[2]

MPI_Send(address, count, datatype, destination, tag, comm)

MPI_Recv(address, maxcount, datatype, source, tag, comm, status)

- Allows maxcount occurrences of items of the form datatype to be received in the buffer starting at address
- source is rank in comm or wildcard MPI_ANY_SOURCE
- tag is an integer used for message matching or wildcard MPI_ANY_TAG
- comm identifies a group of processes and a communication context
- status holds information about the actual message size, source, and tag

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

Combines the elements in the input buffer of each process using the operation op and returns the combined value in the output buffer of the process with rank root

- sendbuf = address of send buffer
- recvbuf = address of receive buffer
- count = number of elements in send buffer
- datatype = data type of elements of send buffer
- op = reduce operation
- root = rank of root process
- comm = communicator

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

- Broadcasts a message from the process with rank root to all other processes of the group.
  - buf = starting address of buffer
  - count = number of entries in buffer
  - datatype = data type of buffer
  - root = rank of broadcast root
  - comm = communicator

Send + receive

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest,
        int sendtag,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, int source,
        int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

- Executes a blocking send and receive operation
- Both send and receive use the same communicator, but possibly different tags
- Semantics as if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads

[1] Es ist ebenfalls nicht erlaubt andere kollektive Oper

[2] Für den Fall das mehr Prozesse verwendet werden, a die Anzahl an betrachteten Punkten erhöt.

Task 2

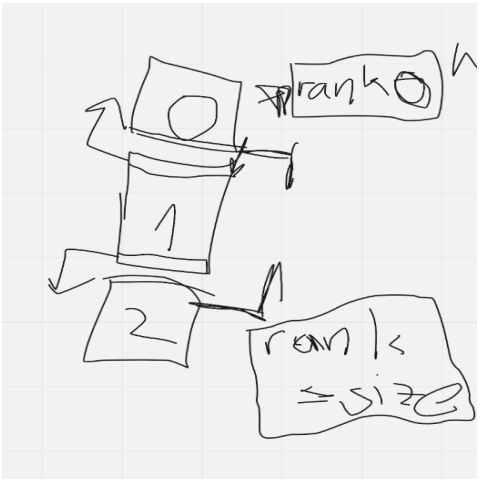
Visualiserung der Parallelisierung (7 Punkte) In dieser Aufgabe verwenden sie Score-P um einen Trace des parallelen Programms zu erzeugen. Dieser kann dann mit dem Visualisierungstool Vampir betrachtet werden.
Um die Visualisierungsdaten zu erzeugen sind folgende Schritte nötig:

- Um Score-P zu verwenden, müssen sie es zunächst mit `module load scorep` auf dem Lichtenberg laden (nur möglich wenn ein Compiler und MPI Modul bereits geladen wurden!).
module load gcc, openmpi, scorep
- Stellen sie dem Compileraufruf zum Kompilieren des Programmes `scorep` voran. (z.B. `mpicc -O2 program.c` zu `scorep mpicc -O2 program.c`) → makefile ändern
- Setzen sie die benötigte Umgebungsvariable mit `export SCOREP_ENABLE_TRACING=true`
- Setzen sie zusätzlich die Umgebungsvariable, die anzeigt, wo die Trace-Daten abgelegt werden sollen.[3]
z.B. mit `export SCOREP_EXPERIMENT_DIRECTORY=$HPC_SCRATCH/<Experimentname>`[4]
- Führen sie das Programm normal mit `mpirun` aus.
mpiexec -n numberofThreads ./integral numberOfPoints
- Dabei wird das angegebene Verzeichnis `$HPC_SCRATCH/<Experimentname>` generiert, in dem die Visualisierungsdaten abgelegt werden.
- Mit dem Visualisierungstool Vampir können sie die Daten visualisieren. Starten sie Vampir dafür mit der traces.otf2 Datei als Argument. (Um Vampir zu benutzen müssen sie X11 forwarding aktiviert haben.[5] Außerdem müssen sie zunächst `module load vampir` ausgeführt haben)

Visualisieren sie ihre Parallelisierung aus Aufgabe 1 mit 8 MPI Prozessen. Beschreiben sie anhand eines Screenshots[6], wie die verschiedenen Teilergebnisse zusammengeführt werden.

missing licence, file

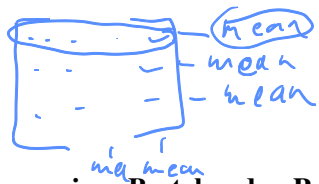
Task 3

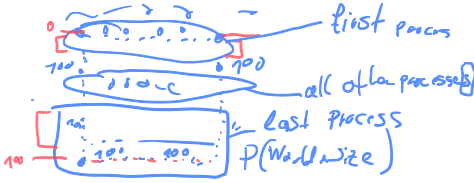


(Serial implementation)

```
//  Determine the new estimate of the solution at the interior points.
//  The new solution W is the average of north, south, east and west
//  neighbors.
//  TODO: Here you may need parts of the matrix that are part of other processes
for (i = 1; i < M - 1; i++)
{
  for (j = 1; j < N - 1; j++)
  {
    // w = (north + south + west + east)/4
    w[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1]) / 4.0;
  }
}
```

[3] Da die erzeugten Daten sehr groß werden können, verwenden sie bitte einen Pfad unter `/work/scratch` und nicht ihr Home verzeichnis.
[4] Die Umgebungsvariable `$HPC_SCRATCH` gibt auf dem Cluster an, wo ihr persönlicher Bereich im scratch Laufwerk ist.
[5] Unter Linux/mac mit der -X option von SSH; Unter Windows z.b. mit bitvise `https://www.bitvise.com/ssh-client`
[6] In Vampir einfach unter dem Menüpunkt Window→Save Screenshot





Task 3

Parallelisierung eines Bestehenden Programms (38 Punkte) Gegeben ist ein sequenzielles C++ Programm in der Datei `heated-plate-parallel.cpp`, dass die Wärmeleitungsgleichung für ein 2-dimensionales Gebiet löst. Nun soll eine Parallelisierung mit MPI erfolgen.

Die Matrix der Temperaturwerte soll zeilenweise über die verschiedenen MPI-Prozesse verteilt werden, sodass jeder Prozess etwa gleich viele Zeilen erhält.[7] Zu keinem Zeitpunkt darf ein Prozess die gesamte Matrix im Speicher halten. Dies wird benötigt, damit es möglich ist Probleme zu lösen, die nicht mehr in den Hauptspeicher eines einzelnen Cluster-Knotens passen.
Beachten Sie dabei, dass man für die Berechnung eines Punktes die Werte der vier benachbarten Punkte benötigt. Hieraus ergibt sich das zu implementierende Kommunikationsschema.

a) **(3 Punkte)** Skizzieren Sie das Kommunikationsschema.

b) **(25 Punkte)** Parallelisieren Sie den bestehenden Code mit Hilfe von MPI.[8] Es ist nicht nötig, die Matrix in irgendeiner Form zusammenzuführen oder auszugeben. Stattdessen soll sie während der gesamten Programmlaufzeit - also auch während der Initialisierung - über die verschiedenen Prozesse verteilt sein.
Verwenden Sie zunächst nur blockierende MPI Operationen. Im Gegensatz zu Aufgabe 1 dürfen Sie kollektive Operationen wie `MPI_Bcast`, `MPI_Reduce` etc. nutzen. Kommentare mit Bearbeitungshinweisen sind im Code mit **TODO** markiert. Time : 1.215

c) **(3 Punkte)** Begründen Sie, warum die Anwendung von nicht-blockierender MPI-Kommunikation in der gegebenen Aufgabenstellung sinnvoll ist und welche Vorteile gegenüber der Verwendung von blockierenden Kommunikationsroutinen entstehen.

Bei blockierenden MPI_Befehlen (MPI_Send, MPI_Recv) wird der restliche code unterhalb des Befehls ERST ausgeführt, wenn der Befehl erfolgreich ist.

d) **(7 Punkte)** Erstellen Sie eine alternative Implementierung des Codes, in der Sie nicht-blockierende MPI Routinen verwenden, um Zeilen der Matrix auszutauschen. Verwenden Sie die nicht-blockierende Kommunikation so, dass dadurch ein Performancegewinn entsteht. Passen Sie das Makefile an, so dass beim Aufruf von `make all` beide Varianten in separaten Executables gebaut werden.

1.64s

Task 4

Vergleich verschiedener Implementationen (12 Punkte) Gegeben sind zwei verschiedene Implementierungen eines parallelen Programms zur Visualisierung der Mandelbrot-Menge. Beide Programme sind korrekt. Vergleichen Sie die beiden Implementationen und diskutieren Sie die Vor- und Nachteile. Gehen Sie dabei auf etwa einer Seite auf folgende Fragen ein:

- Wie werden in Implementation 1 die verschiedenen Rechenaufgaben (Pixel) verteilt?
- Wie werden in Implementation 2 die verschiedenen Rechenaufgaben (Pixel) verteilt?
- Wie wird die Ausgabedatei geschrieben, gibt es hierbei einen Unterscheid zwischen den Implementationen?
- Wie viel Kommunikation erfordern die beiden Implementationen?
- Erläutern Sie mindestens je einen Vor- und einen Nachteil der beiden Implementationen.
- Unter welchen Umständen würden Sie welche Implementation vorziehen?

Hinweis: Die Dateien `utitily.h/.c` werden nur für die Visualisierung benötigt, sind in beiden Fällen gleich und brauchen daher nicht in die Diskussion mit einbezogen zu werden.

[7] Das Programm darf gern abbrechen, falls die Anzahl der Zeilen kleiner die der Prozesse ist.
[8] Das Makefile muss hier so angepasst werden, dass mit `mpic++` kompiliert wird.