TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE
UND HOCHLEISTUNGSRECHNEN
PROF. DR. WOLFGANG E. NAGEL

Komplexpraktikum “Paralleles Rechnen”

Gruppen-Kommunikation

Friedrich Zahn
(Mat.-Nr.: 3966130)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Dr.-Ing. Robert Schöne

Dresden, 24. Februar 2021

Inhaltsverzeichnis

1 Aufgabenstellung 3

2 Benchmark-Umgebung 3
2.1 Software . . . 3
2.2 Hardware . . . 3

3 Lösungsansatz 4

4 Implementation 4
4.1 Bedienungshinweise . . . 4

5 Ergebnisse 5
5.1 Genauigkeit der implementierten numerischen Integrationsmethoden . . . 5
5.2 Skalierungsverhalten der Rechteckregel . . . 6

Literatur 9

A Programmcode 11

1 Aufgabenstellung

Berechnen Sie die Fläche zwischen den Kurven der Funktionen $y^2 = 9x$ und $y = x^2 - 4x + 6$ mittels numerischer Integration!

- Nutzen Sie bei der Programmierung die MPI-Funktionen zur Gruppen-Kommunikation!
- Implementieren Sie als numerisches Integrationsverfahren die „Rechteckregel", und führen Sie folgende Messungen durch:
  - Relativer Fehler bei der Flächenberechnung als Funktion der Anzahl der Integrations-Teilintervalle
  - Geschwindigkeitsgewinn und parallele Effizienz als Funktion der Prozessoranzahl für das Bull HPC Cluster (Taurus)
- Stellen Sie Ihr Programm auf das numerische Integrationsverfahren „Trapezregel" um, und wiederholen Sie Messungen zur Genauigkeit!
- Interpretieren Sie die Ergebnisse!

2 Benchmark-Umgebung

2.1 Software

Die von mir entwickelten Programme wurden auf dem Taurus-Cluster mit den dort vorhandenen Tools kompiliert. Zum Einsatz kamen GCC 7.3.0 mit OpenMPI 3.1.1 und OpenMP 4.5. Die build-Umgebung wurde mit cmake 3.11 generiert.

2.2 Hardware

Alle Messungen wurden auf dem HPC-Cluster „Taurus" des ZIH durchgeführt. Um die Vergleichbarkeit von Messungen sicherzustellen, wurde für alle Benchmarks die „haswell"-Partition genutzt, in welcher alle Nodes über 2 Intel Xeon CPUs, Modell E5-2680 v3, mit jeweils 12 Kernen verfügen[1]. Untereinander sind die Nodes per Infiniband verbunden. Hyperthreading ist auf diesen Nodes deaktiviert, darüber hinaus wurde mittels SLURM die exklusive Nutzung der Nodes angefordert, um Verzerrungen durch anderweitig mitgenutzte Nodes auszuschließen.

3 Lösungsansatz

Zunächst wurde das Problem durch analytische Betrachtung auf das Intervall $[1, 4]$ reduziert, auf welchem die gegebenen Funktionen die gesuchte Fläche entsprechend ihrer Schnittpunkte einschließen. Weiterhin erlaubt dies, lediglich die positiven Funktionswerte der ersten Funktion zu betrachten, und diese somit eineindeutig als einen einzigen C-Funktionsaufruf mit dem Rückgabewert $y = 3\sqrt{x}$ zu modellieren.
Im Übrigen wurden die geforderten numerischen Methoden entsprechend ihrer Wikipedia-Artikel[2, 3] implementiert. Zur Berechnung der eingeschlossenen Fläche müssen diese Methoden lediglich für beide Funktionen innerhalb des obigen Intervalls das Integral liefern, die Subtraktion dieser liefert schließlich das gesuchte Endergebnis.
Da die zu berechnenden Funktionen in das Programm einkompiliert werden konnten, ist im Rahmen der MPI-Gruppenkommunikation lediglich die Aggregation von Teilergebnissen nötig, sofern die Zuordnung der von jedem Prozess auszuführenden Berechnungen anhand inhärent vorhandener Informationen vorgenommen werden kann. Hierfür genügt der MPI-Rank der jeweiligen Prozesse sowie die Kenntnis über die Gesamtzahl der Prozesse. Eine Einschränkung besteht hier insofern, dass die Zahl der Stützintervalle ein Vielfaches der Prozesszahl sein muss.

4 Implementation

Die geforderte numerische Anwendung wurde als C-Programm ausgeführt. Dabei war besondere Sorgfalt beim Aufspannen der benötigten Stützstellen von Nöten, da der verwendete `double` Datentyp nur begrenzte Präzision beim Darstellen von Fließkommazahlen liefern kann. Insofern ist eine exakte äquidistante Aufteilung des zu integrierenden Intervalls in eine beliebige Zahl Stützstellen schon rein numerisch nicht möglich. Diesem Problem wurde begegnet, indem stets die tatsächliche Distanz zwischen Stützstellen zur Berechnung der Rechteck- bzw. Trapezfläche genutzt wurde, nicht etwa die Ideallänge die sich aus der Intervalllänge geteilt durch die Zahl der Stützintervalle ergäbe. Darüber hinaus wurde als letzte Stützstelle stets der angeforderte rechte Rand des Intervalls eingesetzt, nicht die sich aus einer Fortsetzung von der vorletzten Stützstelle rechnerisch ergebende.
Weitere Optimierung der hier gewählten Implementierung wäre für den Einsatz mit einer extrem hohen Zahl von Stützstellen von Nöten, da in diesem Fall sehr nah beieinanderliegende Fließkommazahlen voneinander subtrahiert würden, was zu einem enormen Präzisionsverlust führen kann[5]. Auch die Aggregation der Teilergebnisse innerhalb eines Prozesse wäre zu überarbeiten, um die Addition von sehr kleinen Zahlen zum wesentlich größeren Teilergebnis zu vermeiden. Im Rahmen der hier beispielhaft betrachteten Funktionen ist derartige Präzision jedoch nicht notwendig, da bereits mit einer weit niedrigeren Zahl von Stüzstellen sehr akurate Ergebnisse ermittelt werden können.
Die geforderte Gruppen-Kommunikation zwischen den MPI-Prozessen wurde mittels zwei Aufrufen der Routine `MPI_Reduce` umgesetzt, die MPI-Standardoperation `MPI_SUM` aggregiert dabei die Teilergebnisse aus jedem Prozess in eine Summe pro Funktion.
Der gesamte Programmcode ist im Anhang als Listing 1 zu finden.

4.1 Bedienungshinweise

Das Programm erwartet die Übergabe zweier Parameter, an erster Stelle die gewünschte Zahl der zu nutzenden Stützintervalle, sowie an zweiter Stelle die gewünschte numerische Methode - eine `0` für die Rechteckmethode oder eine `1` für die Trapezmethode.
Das Programm erkennt selbständig, wie viele Prozesse im MPI-Kontext zur Verfügung stehen, und verteilt die Arbeitslast gleichmäßig, wofür die Zahl der geforderten Stützintervalle stets auf ein Vielfaches der Prozessorenzahl abgerundet wird.
Die Ergebnisse werden an eine Datei mit dem Namen `kppr_2.out` angefügt, jede Zeile enthält dann enstprechend folgende Informationen aus einem Aufruf: Prozessorenzahl, tatsächliche Stützintervallzahl, absoluter Fehler, relativer Fehler in Prozent, Gesamtlaufzeit.

5 Ergebnisse

5.1 Genauigkeit der implementierten numerischen Integrationsmethoden

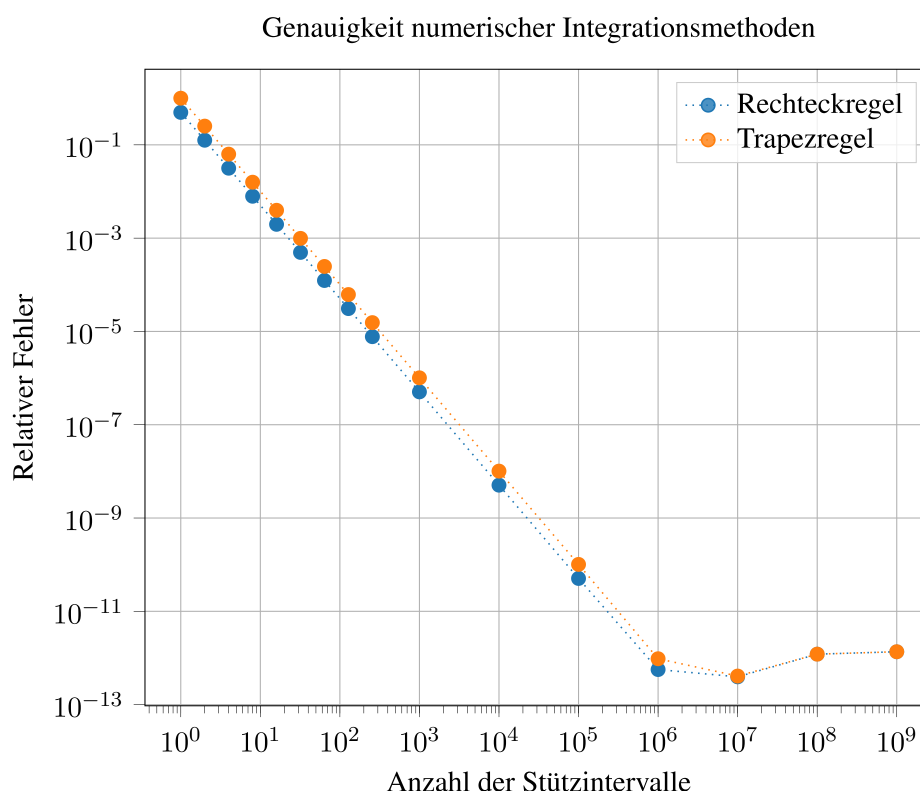

Abbildung 1: Der relative Fehler $\frac{|A-A_{\text{Num}}|}{A}$ bei der numerischen Berechnung der eingeschlossenen Fläche $A = \left|\int_1^4 (x^2 - 4x + 6)\mathrm{d}x - \int_1^4 3\sqrt{x}\mathrm{d}x\right|$ für die beiden numerischen Integrationsmethoden in Abhängigkeit von der Zahl der Stützintervalle, über die genähert wurde.

Der Verlauf der relativen Fehler für die beiden implementierten Integrationsmethoden ist in Abbildung 1 dargestellt. In der doppelt logarithmischen Darstellung ist ersichtlich, dass die Fehler zunächst gleichmäßig mit steigender Zahl der Näherungsintervalle sinken. Der gradlinige Verlauf legt nahe, dass es sich bei der Abhängigkeit des relativen Fehlers um eine strikte Potenzfunktion ($c \cdot x^p$) handelt.
Dieses Verhalten dauert bis zur Nutzung von 10^6 und mehr Stützintervallen an, und der relative Fehler stagniert fortan bei einem Wert von circa 10^{-12}, was angesichts des zugrundeliegenden Datentyps (`double`, 64 bit nach IEEE754) im Bereich der maximal möglichen Darstellungspräzision liegt. Eine feinere Auflösung kann also nicht mehr zu einem genaueren Ergebnis führen, da die Genauigkeitsgewinne in der numerischen Methode gegenüber Darstellungsverlusten und Auslöschungseffekten in der Fließkommadarstellung und -arithmetik nicht mehr ins Gewicht fallen.
Es ist weiterhin auffällig, dass die Trapezmethode einen durchgehend höheren Fehler aufweist, als die Rechteckmethode. Dies entspricht nicht unbedingt der intuitiven Erwartung, die Trapezmethode scheint aufwändiger und nähert rein optisch den Verlauf der Kurven besser an. Jedoch sind beide Methoden ihrer Natur nach nur geeignet, affin-lineare Funktionen exakt abzubilden und weisen dementsprechend ähnliches Residual-Verhalten auf. Berechnet man den Fehler auf den Methoden analytisch, ergibt sich zwischen den beiden Methoden ein Faktor von genau 2[4].
Setzt man die relativen Fehler der beiden Methoden ins Verhältnis, wie in Abbildung 2, bestätigt sich diese theoretische Überlegung. Bis zum Erreichen des Darstellungslimits ist der relative Fehler auf der Trapezmethode fast genau zwei mal so groß wie der auf der Rechteckmethode. Im Bereich stagnierender Genauigkeit verschwindet dieser Unterschied, und beide Methoden liefern praktisch das gleich Ergebnis,

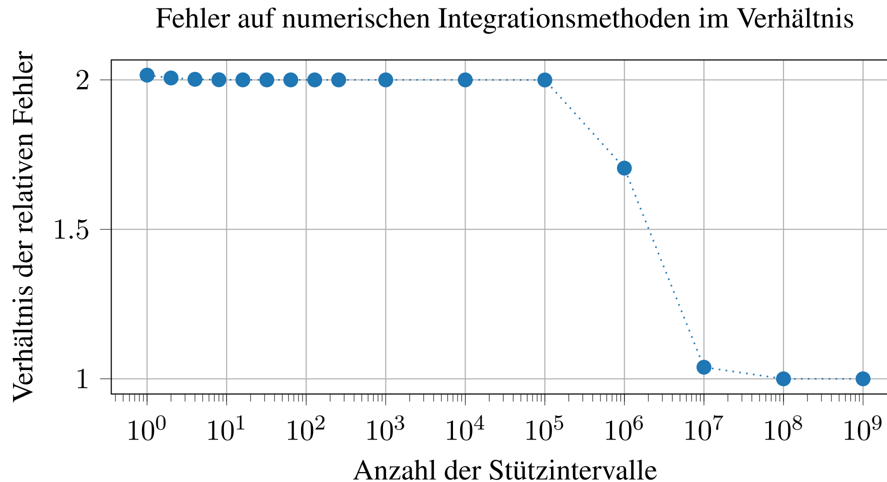

Abbildung 2: Die relativen Fehler der Trapez- und Rechteckmethode ins Verhältnis gesetzt, $\left|\frac{A-A_{\text{Trapez}}}{A-A_{\text{Rechteck}}}\right|$, in Abhängigkeit von der Zahl der Stützintervalle, über die genähert wurde.

wobei die Trapezmethode etwas mehr Stützstellen zum Erreichen des Darstellungslimits von 10^{-12} benötigt.

5.2 Skalierungsverhalten der Rechteckregel

Im Weiteren wurde das Skalierungsverhalten der Rechteckregel bei der Parallelisierung des Algorithmus untersucht. Bei der Verwendung von p Prozessen und damit Cores wurde jedem Prozess $\frac{1}{p}$ des Gesamtintervalls und ebenso $\frac{1}{p}$ der gesamt zu nutzenden Stützintervallzahl zugewiesen.
Um eine Arbeitslast zu simulieren, bei der eine Parallelisierung überhaupt relevante Zeitersparnisse bietet, wurde die ansonsten im Bezug zur bereits betrachteten erreichbaren Genauigkeit sehr hohe Zahl von $960 \cdot 10^6$ Stützintervallen gewählt, welche außerdem über die nötigen Teiler verfügt, sodass eine Aufteilung der Last auf eine beliebige Zahl von Nodes mit je 24 Cores möglich ist. Es wird also das Verhalten im Sinne des „strong scaling“ betrachtet.
In Abbildung 3 ist der erreichte Speedup in Abhängigkeit von der Zahl der genutzten Cores dargestellt. Bis zu einer Zahl von 24 Cores, also einer kompletten Node, werden durchgehend Speedups nahe des theoretisch möglichen erreicht, die Kurve liegt nur knapp unterhalb der ebenfalls eingezeichneten Ideallinie eines perfekt parallelisierbaren Problems. Mit dem weiteren Einsatz von mehr als 24 Cores zeichnet sich der Beginn der „diminishing returns“ ab, die Kurve knickt ab und bleibt zunehmend hinter der Ideallinie zurück. Ab einer Zahl von 192 Cores stagniert der Verlauf schließlich fast völlig, ein Einsatz weiterer Ressourcen liefert keine weitere Beschleunigung mehr.
Dieses Verhalten zeigt sich noch deutlicher, wenn man die selben Messdaten hinsichtlich ihrer Effizienz betrachtet und wie in Abbildung 4 darstellt. Der Wert der Effizienz gibt hier an, welchen rechnerischen Anteil an der Laufzeit - für die die angeforderten Ressourcen blockiert und nicht anders genutzt werden können - jeder Core effektiv tatsächlich Arbeit verrichtet, im Vergleich zur rein sequentiellen Ausführung auf einem Core.
Bei der Verwendung von 24 Cores wird noch eine annehmbare Effizienz von etwas über 80% erreicht, jenseits dessen sinkt die Effizienz jedoch fast gradlinig ab und geht für eine Zahl von Cores jenseits der 1000 effektiv gegen 0, da wie bereits ausgeführt eine weitere Hinzunahme von Ressource keinen weiteren Speedup mehr bringt, es wird lediglich entsprechend mehr Prozessoren-Zeit verschwendet.

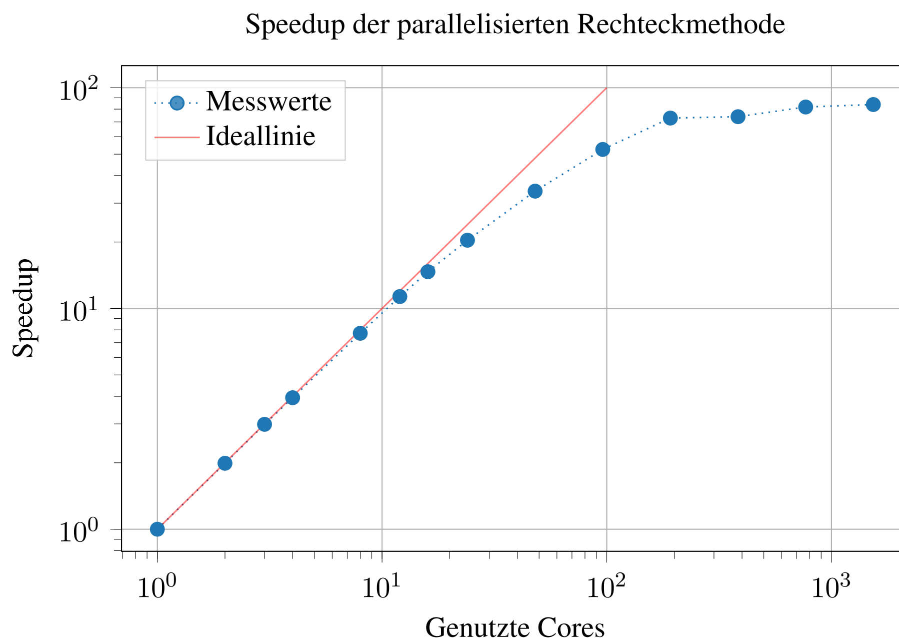

Abbildung 3: Speedup für die Implementation der Rechteckregel in Abhängigkeit von der Zahl der genutzten Cores. Die rote Gerade stellt den Kurvenverlauf für ein ideal parallelisierbares Problem da, bei dem sämtliche Berechnungen ohne Overhead oder sequentielle Anteile gleichmäßig auf alle verfügbaren Cores verteilt werden können. Gemessen wurden die Laufzeiten zur numerischen Integration über $960 \cdot 10^6$ Stützintervalle.

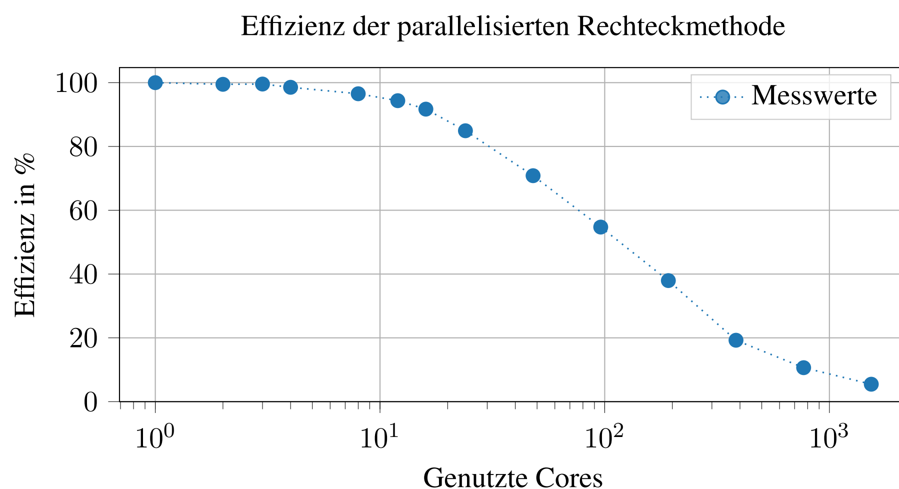

Abbildung 4: Effizienz der parallelisierten Ausführung der Rechteckregel, dargestellt ist der Anteil der Laufzeit der sequentiellen Ausführung gegenüber der Gesamt-Laufzeit des Programms über alle genutzten Cores aufsummiert. Gemessen wurden die Laufzeiten zur numerischen Integration über $960 \cdot 10^6$ Stützintervalle.

Literatur

[1] *Hardware-Informationen zum Taurus-Cluster im ZIH-Kompendium*. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus, . – [Online; Stand 18. Januar 2021]

[2] WIKIPEDIA: *Mittelpunktsregel — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Mittelpunktsregel&oldid=188992583, 2019. – [Online; Stand 18. Januar 2021]

[3] WIKIPEDIA: *Trapezregel — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Trapezregel&oldid=197761406, 2020. – [Online; Stand 18. Januar 2021]

[4] WIKIPEDIA CONTRIBUTORS: *Riemann sum — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Riemann_sum&oldid=995959887, 2020. – [Online; Stand 19. Januar 2021]

[5] WIKIPEDIA CONTRIBUTORS: *Floating-point arithmetic — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Floating-point_arithmetic&oldid=997728268, 2021. – [Online; Stand 18. Januar 2021]

A Programmcode

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>

double f1(double x){
    return x*x - 4.0*x + 6.0;
}

double f2(double x){
    // Due to how f1 and f2 intersect, we only need the positive branch.
    return 3.0*sqrt(x);
}

double F1(double x){
    return x*x*x/3.0 - 2.0*x*x + 6.0*x;
}

double F2(double x){
    return 2.0 * pow(x, 3.0/2.0);
}

double midpointInt(double (*fPtr)(double), double xLeftStop, double xRightStop, int
    samples){
    double idealLength = (xRightStop - xLeftStop) / samples;

    double A = 0.0;
    double xLeft = xLeftStop;
    double xRight = xLeft + idealLength;
    double x;
    double length;
    int i;

    for(i = 0; i<samples - 1; i++){
        length = xRight - xLeft;
        x = xLeft + 0.5*length;
        A += length * (*fPtr)(x);
        xLeft = xRight;
        xRight = xLeft + idealLength;
    }

    // Account for floating point inaccuracy, make sure we don't over/undershoot
    with the last sample.
    xRight = xRightStop;
    length = xRight - xLeft;
    x = xLeft + 0.5*length;
    A += length * (*fPtr)(x);

    return A;
}


double trapezoidalInt(double (*fPtr)(double), double xLeftStop, double xRightStop,
    int samples){
    double idealLength = (xRightStop - xLeftStop) / samples;

    double A = 0.0;
    double xLeft = xLeftStop;
    double xRight = xLeft + idealLength;
```

```
58     double length;
59     int i;
60 
61     for(i = 0; i<samples - 1; i++){
62         length = xRight - xLeft;
63         A += length * ((*fPtr)(xLeft) + (*fPtr)(xRight)) * 0.5;
64         xLeft = xRight;
65         xRight = xLeft + idealLength;
66     }
67 
68     // Account for floating point inaccuracy, make sure we don't over/undershoot
       with the last sample.
69     xRight = xRightStop;
70     length = xRight - xLeft;
71     A += length * ((*fPtr)(xLeft) + (*fPtr)(xRight)) * 0.5;
72 
73     return A;
74 }
75 
76 int main(int argc, char** argv) {
77     // Initialize the MPI environment
78     MPI_Init(NULL, NULL);
79 
80     // Get the number of processes
81     int world_size;
82     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
83 
84     // Get the rank of the process
85     int world_rank;
86     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
87 
88     int samplesGlobal =  atoi(argv[1]);
89     int intFlag = atoi(argv[2]);
90     double (*intPtr)(double (*)(double), double, double, int);
91     if(intFlag == 0){
92         intPtr = &midpointInt;
93     } else if(intFlag == 1){
94         intPtr = &trapezoidalInt;
95     } else {
96         printf("Unkown numerical integration method requested.\n");
97         return EXIT_FAILURE;
98     }
99 
100     struct timespec tStart, tEnd;
101     if(world_rank == 0){
102         clock_gettime(CLOCK_MONOTONIC, &tStart);
103     }
104 
105     // f1 and f2 intersect here:
106     double xLeftStopGlobal = 1.0;
107     double xRightStopGlobal = 4.0;
108 
109     int samples = samplesGlobal / world_size;
110 
111     double sectionLength = (xRightStopGlobal - xLeftStopGlobal) / world_size;
112     double xLeftStop = xLeftStopGlobal + world_rank * sectionLength;
113     double xRightStop;
114     if(world_rank != world_size - 1){
115         xRightStop = xLeftStop + sectionLength;
116     } else {
117         xRightStop = xRightStopGlobal;
118     }
119 
```

```
120     double A1Local = (*intPtr)(&f1, xLeftStop, xRightStop, samples);
121     double A2Local = (*intPtr)(&f2, xLeftStop, xRightStop, samples);
122 
123     double A1, A2;
124     MPI_Reduce(&A1Local, &A1, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
125     MPI_Reduce(&A2Local, &A2, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
126 
127     if(world_rank == 0) {
128         double A = A2 - A1;
129         clock_gettime(CLOCK_MONOTONIC, &tEnd);
130 
131         double timeTaken = tEnd.tv_sec - tStart.tv_sec + (tEnd.tv_nsec - tStart.
    tv_nsec) * 1e-9; // in seconds
132         printf("Enclosed area is %f\n", A);
133 
134         double A1True = F1(xRightStopGlobal) - F1(xLeftStopGlobal);
135         double A2True = F2(xRightStopGlobal) - F2(xLeftStopGlobal);
136         double ATrue = A2True - A1True;
137         printf("Result differs from analytical calculation by %f, "
138                "which is %f Percent.\n", fabs(ATrue - A), fabs(100 * (ATrue - A) /
    ATrue));
139         FILE *outf;
140         outf = fopen("kppr_2.out", "a");
141         fprintf(outf, "%d, %d, %.16f, %.16f, %.16f\n",
142                 world_size, samples*world_size, fabs(ATrue - A), fabs(100 * (ATrue -
     A) / ATrue), timeTaken);
143         fclose(outf);
144     }
145 
146     MPI_Finalize();
147     return EXIT_SUCCESS;
148 }
```

Listing 1: Der verwendete Programmcode für die durchgeführten Messungen.