TECHNISCHE UNIVERSITÄT DRESDEN

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

# Komplexpraktikum “Paralleles Rechnen”

# Punkt-zu-Punkt-Kommunikation mit MPI

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 Implementierung** **4**

**5 Ergebnisse** **5**
5.1 Übertragungsverhalten innerhalb eines Knotens . . . . . 5
5.2 Übertragungsverhalten zwischen zwei Knoten . . . . . 8

**Literatur** **11**

**A Programmcode** **13**

# 1 Aufgabenstellung

Schreiben Sie auf der Basis der MPI-Funktionen zur Punkt-zu-Punkt-Kommunikation ein Programm zur Ermittlung der Gesamtlatenz und der Kommunikationsbandbreite!

- Ermitteln Sie die gemessene Zeit für eine Punkt-zu-Punkt-Kommunikation in Abhängigkeit von der Nachrichtenlänge! Berechnen Sie dabei für jede Nachrichtenlänge die durchschnittliche Zeit aus 100 aufeinanderfolgenden Kommunikationen!
- Stellen Sie die übertragenen Daten pro Zeiteinheit als Funktion der Nachrichtenlänge dar! Wie groß ist die Kommunikationsbandbreite?
- Führen Sie die Messungen auf dem Bull HPC Cluster (Taurus) sowohl lokal (zwei Cores auf einem Knoten) als auch remote (jeweils ein Core auf zwei Knoten) durch!
- 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

Eine direkte Messung der Laufzeit zwischen zwei MPI-Prozessen war nicht möglich, da es keine triviale Möglichkeit gibt, die Uhren zwischen Prozessen auf unterschiedlichen Knoten mit hinreichender Genauigkeit zu synchronisieren, insbesondere da die Laufzeit einer solchen Synchronisationsmitteilung a priori nicht bekannt ist. Daher wurde die Umlaufzeit für die Übermittlung einer bestimmten Datenmenge $D$ gemessen, also die Zeit die vergeht, während Prozess 0 die Daten sendet, Prozess 1 sie empfängt, sie wiederum sendet und schließlich Prozess 0 sie wieder empfangen hat, was im Folgenden auch kurz als RTT (Round Trip Time) bezeichnet wird.
Die daraus resultierende Bandbreite wurde als $\frac{2 \cdot D}{T_{\mathrm{RTT}}}$ berechnet.
Die Spanne und Auflösung der gewählten Datenmengen basiert auf grundlegenenden Überlegungen zur zu Grunde liegenden Architektur. Es ist zu erwarten, dass insbesondere im Bereich bis hin zu einigen kiB Quantisierungsartefakte in den Messungen auftreten, da diverse Puffergrößen, Paketlimits und Verwaltungseinheiten in diesem Bereich liegen.

## 4 Implementierung

Der Programmcode wurde in C geschrieben, und verwendet eine simple Schleifenkonstruktion, um in der äußeren Schleife die gewünschten Datenmengen zu spezifizieren, und in der inneren Schleife die geforderten 100 aufeinanderfolgenden Kommunikationen auszuführen.

```
data = malloc(len);

... // MPI- und Speicher-Warmup

clock_gettime(CLOCK_MONOTONIC, &tStart);

for (int r = 0; r < REPS; r++) {

    if (world_rank == 0) {
        MPI_Send(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD, NULL);
    }

    if (world_rank == 1) {
        MPI_Send(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD, NULL);

    }
}
clock_gettime(CLOCK_MONOTONIC, &tEnd);

free(data);
```

Listing 1: Der C-Code der implementierten Benchmark-Schleife.

Die Punk-zu-Punkt-Kommunikation wurde dabei mittels `MPI_Send` und `MPI_Recv` implementiert, wie in Listing 1 aufgeführt. Die Zeitmessung erfolgte über die C-Routine `clock_gettime(CLOCK_MONOTONIC, ...)` die eine Auflösung von Nanosekunden liefert und damit wesentlich genauer misst, als die zu erwartenden Laufzeiten im Mikrosekunden-Bereich.
Der vollständige Programmcode befindet sich im Anhang als Listing 2.

# 5 Ergebnisse

Je nach Auflösung und Spanne an Datenmengen, ergeben sich unterschiedliche Verhalten für die Umlaufzeiten und die damit invers einhergehende Bandbreite, weshalb ich mich im Folgenden entschieden habe, diese jeweils getrennt darzustellen.

## 5.1 Übertragungsverhalten innerhalb eines Knotens

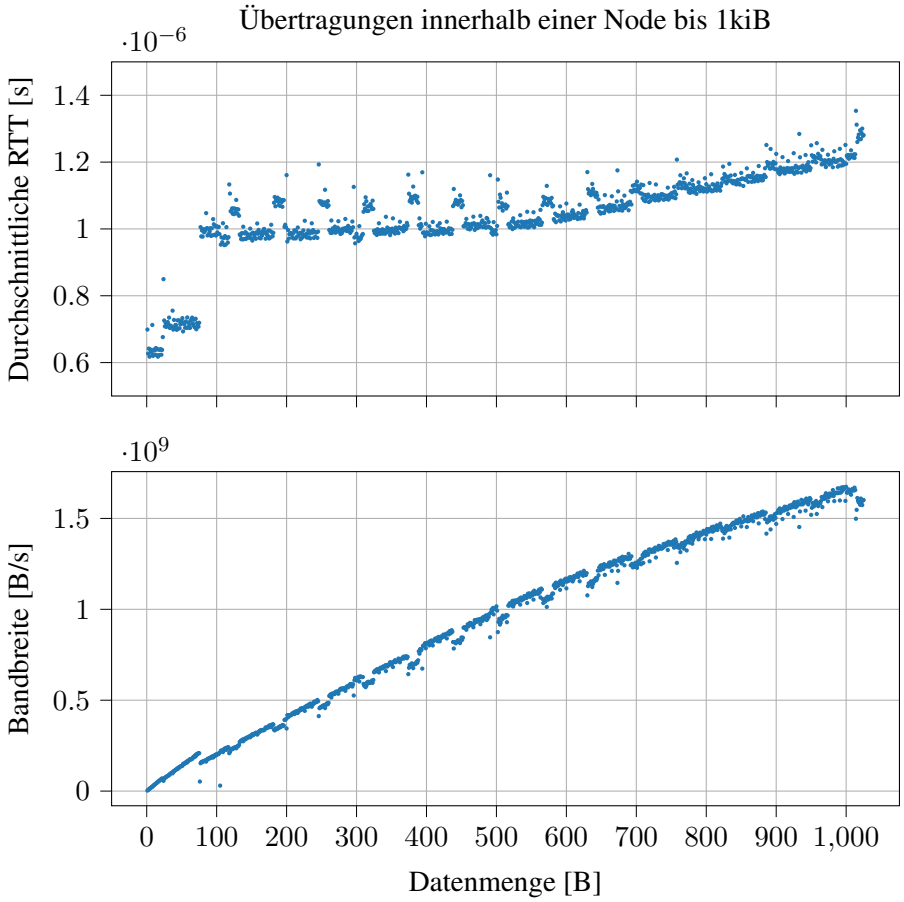


Abbildung 1: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen innerhalb einer Node, bis zu einer Datenmenge von einem Kibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Dargestellt sind alle 1024 Messpunkte.

In Abbildung 1 ist ersichtlich, dass bei der MPI-Kommunikation auf Basis von shared memory die konkrete Speicherarchitektur beim Übertragen kleiner Datenmengen einen erheblichen Einfluss hat. Besonders hevorstechende Unstetigkeiten im Verlauf der Umlaufzeiten treten beim Übertragen von 24 Byte sowie 76 Byte (was 24 Byte + 32 Byte entspricht) auf, was nahe legt, dass die genutzte MPI-Bibliothek diese kleinsten Datenmengen noch mithilfe von Registern o.ä. überträgt, und ab der Verwendung von circa 8 Wortlängen an Daten zur Nutzung regulären Speichers wechselt. Im Weiteren erhöht sich in regelmäßigen Abständen von 64 Byte, was der Länge einer Cachezeile in den genutzten CPUs entspricht[3], die Umlaufzeit sprunghaft. Ein entgegengesetzter Sprung erfolgt meist 15 Byte später. Möglicherweise liegt hier eine Verwendung von einer Kombination aus heterogenen Speichern vor, die in bestimmten Kombinationen zu erhöhten Latenzen führt.
In Abbildung 2 setzt sich der generelle Trend linear anwachsender Umlaufzeiten fort, größere Unstetigkeiten treten hier im Bereich von Vielfachen der üblichen Pagesize auf, insbesondere bei 8 Kibibyte und dann noch einmal bei 16 Kibibyte. Ab letzter Datenmenge sackt die Bandbreite auch nachhaltig erheblich

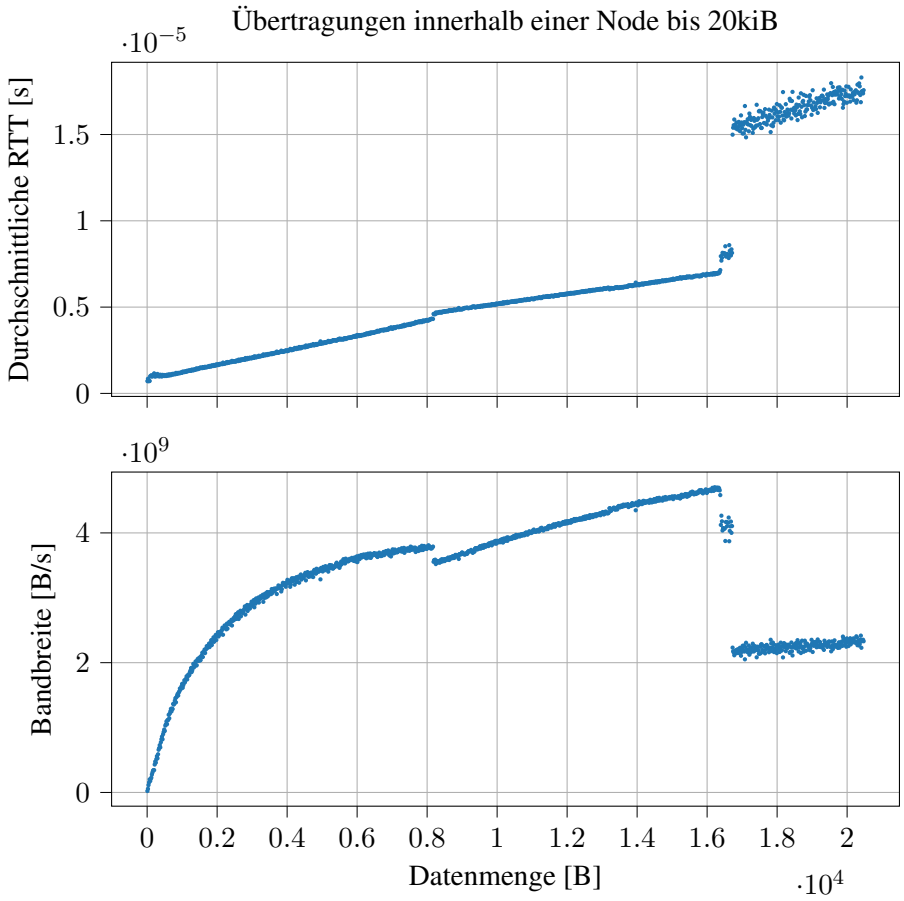


Abbildung 2: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen innerhalb einer Node, bis zu einer Datenmenge von 20 Kibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Dargestellt ist jeder 16. Messpunkt, also $n \cdot 16$ B.

ab, was vermuten lässt, dass hier erneut ein Wechsel des zugrundeliegenden Speicherverhaltens stattfindet, im Weiteren weisen die Messungen auch deutlich höheres Rauschen auf.
Abschließend wurden Messungen zum Skalierungsverhalten für größere Datenmengen durchgeführt, diese sind in Abbildung 3 dargestellt. Es ist ersichtlich, dass mit steigender Datenmenge die Bandbreite zunächst immer weiter ansteigt, ein prinzipieller Trend der auch schon bei den kleineren Datenmengen zu beobachten war. Mit steigender Datenmenge fällt der Anteil volumenunabhängigen Overheads und statischer Latenzen zunehmend weniger ins Gewicht, auch können Datenbusse besser ausgelastet werden, sodass die effektive Bandbreite steigt. Dieser Trend hält bis zu einer Datenmenge von einem Mebibyte an, und stagniert im Weiteren knapp unterhalb eines Werts von 4 Gibibyte pro Sekunde, deutlich unterhalb des theoretischen CPU-Maximums von 68 GB/s, welches Intel für die Speicherbandbreite angibt[2]. Dies ist insofern erklärbar, dass im hier verwendeten Benchmark bestenfalls 2 der 12 Cores gleichzeitig auf den Speicher zugreifen, und dementsprechend nur ein Bruchteil der Speicherkanäle tatsächlich genutzt werden kann.

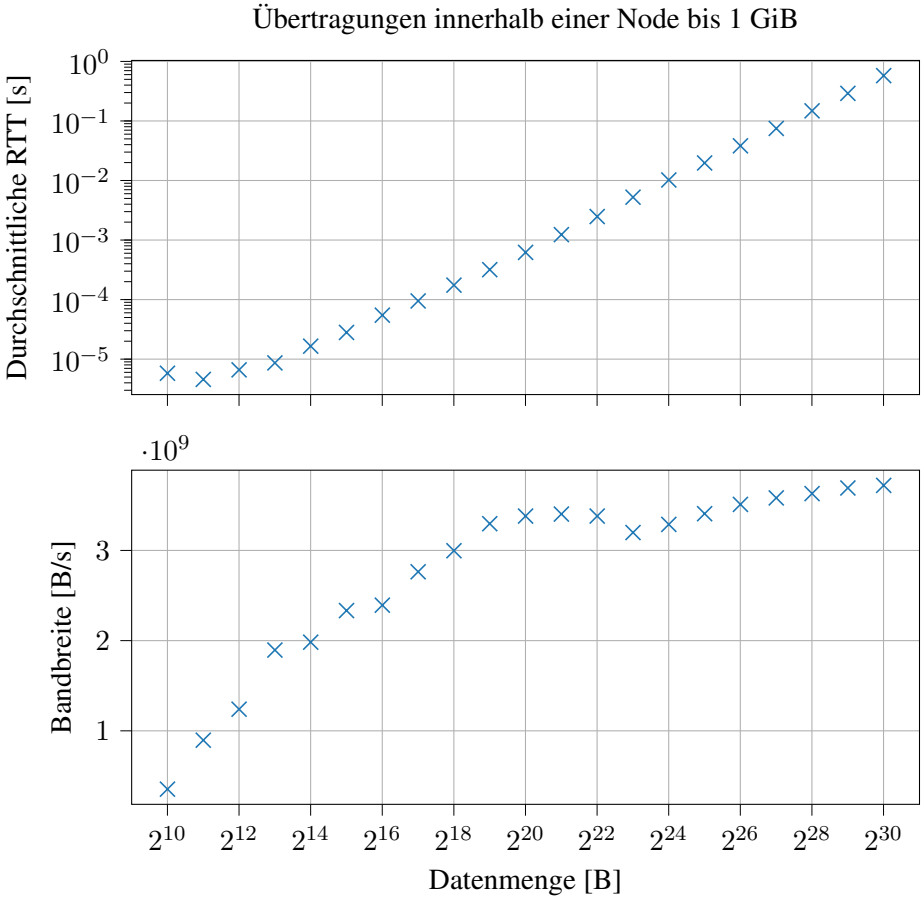


Abbildung 3: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen innerhalb einer Node, bis zu einer Datenmenge von 1 Gibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Gemessen und dargestellt ist jede volle Zweierpotenz ab 1024 B.

### 5.2 Übertragungsverhalten zwischen zwei Knoten

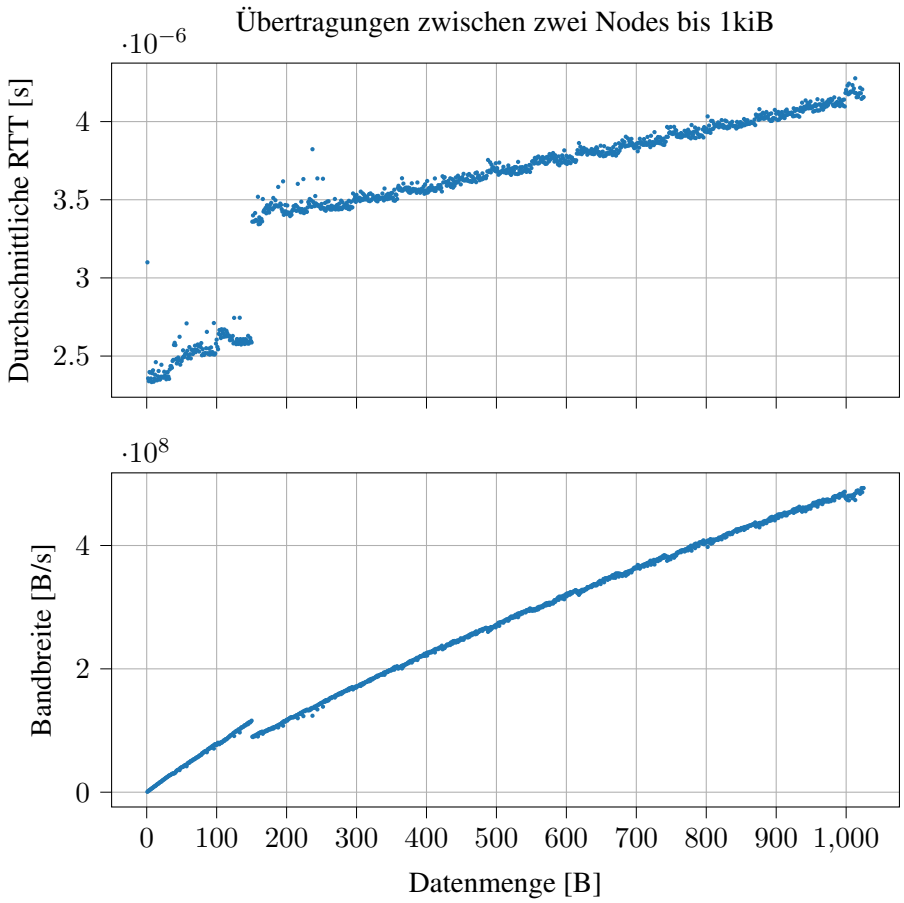


Abbildung 4: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen zwischen zwei Nodes, bis zu einer Datenmenge von einem Kibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Dargestellt sind alle 1024 Messpunkte.

Die Messergebnisse für kleinste übertragene Datenmengen zwischen zwei räumlich getrennten Nodes, also unter Nutzung der Infiniband-Netzwerktechnologie, sind in Abbildung 4 dargestellt. Es sind prinzipiell ähnliche Artefakte wie in Abbildung 1 erkennbar, jedoch fallen diese aufgrund der höheren Grund-Latenz, die durch die Netzwerkkommunikation zustande kommt, wesentlich weniger stark ins Gewicht. Die Differenz beträgt schon für die kleinsten Datenmengen rund $1,8$ Mikrosekunden, erhöht sich also auf ein Vierfaches von $0,6$ auf $2,4$ Mikrosekunden.
Ein herausstechendes Verhalten ist jedoch der sprunghafte Anstieg der Umlaufzeit beim Übermitteln von 151 Byte und mehr. Dieses Verhalten korreliert mit keinem mir bekannten Puffer, auch dürfte es weit unterhalb etwaiger Netzwerklimitationen liegen.
Betrachtet man die in Abbildung 5 dargestellten Messungen für Datenmengen bis 20 Kibibyte, zeigt sich ein deutlich gleichmäßigeres Verhalten als in der äquivalenten Abbildung 2, die Kurve ist aufgrund der höheren Grund-Latenz insbesondere im Bereich über 16 Kibibyte geglättet. Kleinere Unstetigkeiten zeigen sich wieder beim Überschreiten von Vielfachen der Pagesize, bzw. möglicherweise der Infiniband-Paketgröße, von 4 Kibibyte. Es ist anzumerken, dass insbesondere kein Sprung im Bereich oder mit einer Periode von 1500 Byte, der üblichen Ethernet-MTU, auftritt, da im Taurus-Cluster Infiniband zum Einsatz kommt.
Die Bandbreitenmessungen für große Datenmengen zwischen zwei Nodes, wie in Abbildung 6 dargestellt, weisen einige deutliche Unterschiede gegenüber dem Übertragungsverhalten innerhalb einer Node auf. Auffallend ist zunächst die Unstetigkeit, die beim Übertragen von mehr als $2^{24}$ Byte (16 Mebibyte)

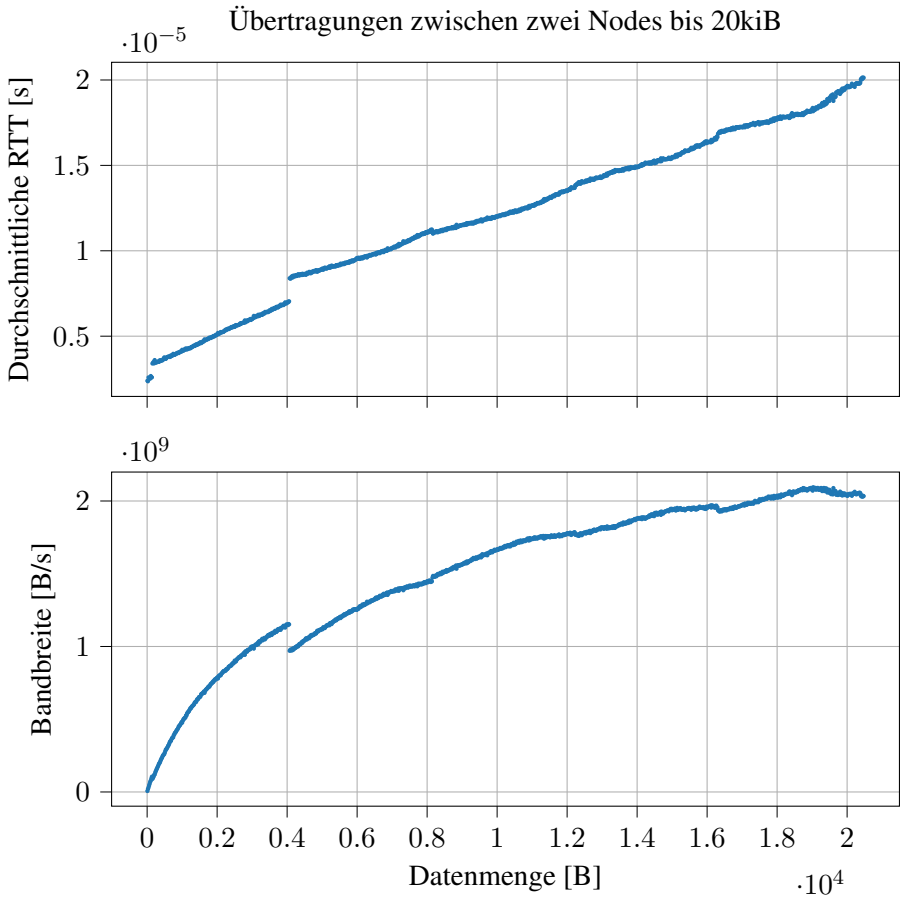


Abbildung 5: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen zwischen zwei Nodes, bis zu einer Datenmenge von 20 Kibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Dargestellt ist jeder 16. Messpunkt, also $n \cdot 16$ B.

auftritt. Eine mögliche Erklärung wäre, das hier das Fassungsvermögen des L3-Caches der CPUs (30 MiB) überschritten wird. Da beim Übertragen innerhalb einer Node jedoch kein solcher Einbruch zu beobachten ist, scheint dies nicht der kausale Zusammenhang zu sein. Eine andere Möglichkeit wäre, dass die Puffer der Infiniband-Netzwerkkarten oder ähnlicher beteiligter Komponenten nur 16 MiB fassen.
Generell ist bemerkenswert, dass insbesondere für Datenmengen größer als $2^{18}$ Byte durchweg höhere Übertragungsraten erreicht werden, als innerhalb einer Node gemessen wurden. Möglicherweise ist der direkte Speicherzugriff über Infiniband hier tatsächlich vorteilhafter, als der - unter Umständen konkurrierende - Zugriff von zwei Prozessen auf den selben Hauptspeicher.

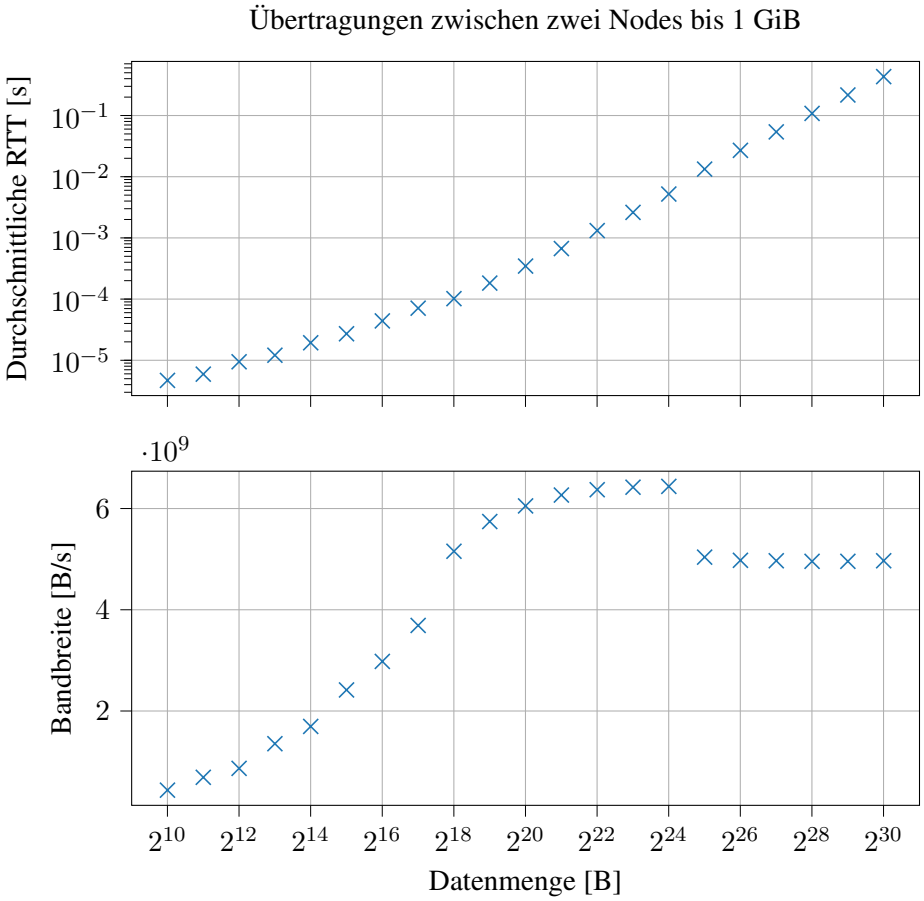


Abbildung 6: Umlaufzeiten und die sich daraus ergebende Bandbreite für Übertragungen zwischen zwei Nodes, bis zu einer Datenmenge von 1 Gibibyte. Alle Übertragungen wurden 100 mal ausgeführt und darüber gemittelt. Gemessen und dargestellt ist jede volle Zweierpotenz ab 1024 B.

## 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] INTEL CORPORATION: *Intel Ark Dokumentation zum Prozessor E5-2680V3*. https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e5-2680-v3-30m-cache-2-50-ghz.html, . – [Online; Stand 20. Januar 2021]

[3] WIKIPEDIA CONTRIBUTORS: *Haswell (microarchitecture) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Haswell_(microarchitecture)&oldid=998401162, 2021. – [Online; Stand 18. Januar 2021]

## A Programmcode

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

#define REPS 100

int main() {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    FILE *outf;
    if (world_rank == 0) {
        outf = fopen("kppr_1.out", "w");
    }

    struct timespec tStart, tEnd;
    void *data;

    //for(int len=1; len < 20481; len++) {
    for(int len=1024; len<=1<<30; len*=2){
        double timeTaken = 0;

        data = malloc(len);

        // Fire once to ensure comms are established
        if (world_rank == 0) {
            MPI_Send(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        } else {
            MPI_Recv(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD, NULL);
        }

        if (world_rank == 1) {
            MPI_Send(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        } else {
            MPI_Recv(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD, NULL);
        }

        clock_gettime(CLOCK_MONOTONIC, &tStart);

        for (int r = 0; r < REPS; r++) {

            if (world_rank == 0) {
```

```
                MPI_Send(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
            } else {
                MPI_Recv(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD, NULL);
            }

            if (world_rank == 1) {
                MPI_Send(data, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
            } else {
                MPI_Recv(data, len, MPI_CHAR, 1, 0, MPI_COMM_WORLD, NULL);

            }
        }
        clock_gettime(CLOCK_MONOTONIC, &tEnd);
        timeTaken += tEnd.tv_sec - tStart.tv_sec
                     + (tEnd.tv_nsec - tStart.tv_nsec) * 1e-9;

        free(data);

        if (world_rank == 0){
            timeTaken = timeTaken / REPS;
            fprintf(outf, "%d, %.12f\n", len, timeTaken);
        }
    }

    if (world_rank == 0) fclose(outf);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Listing 2: Der verwendete Programmcode für die durchgeführten Messungen.