TECHNISCHE UNIVERSITÄT DRESDEN

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

Komplexpraktikum “Paralleles Rechnen”

Abgeleitete Datentypen

Friedrich Zahn
(Mat.-Nr.: 3966130)

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

Dresden, 19. März 2021

Inhaltsverzeichnis

1 Aufgabenstellung

Gegeben ist eine Matrix A mit m Zeilen und n Spalten sowie ein Vektor B mit n Elementen. Gesucht ist das Punktprodukt zwischen jeder Matrixzeile und dem Vektor.

- Es ist ein Programm für variable Problemgröße und variable Prozessoranzahl p unter Nutzung der MPI-Funktionen für abgeleitete Datentypen zu schreiben. Die Matrix soll durch den Masterprozessor in p Subarrays zerlegt und auf alle vorhandenen Prozessoren verteilt werden, wobei jeder Prozessor die Punktprodukte für ein Subarray berechnet. Die Ergebnisse werden durch den Masterprozessor eingesammelt und zusammen mit den Eingangsdaten ausgegeben. Dabei werden die Subarrays mit `MPI_Send(..)` und `MPI_Recv(..)` verteilt und eingesammelt.
- Messen Sie die Zeitdauer der Berechnungen und stellen Sie den Geschwindigkeitsgewinn S_p und die parallele Effizienz E_p für drei geeignete Problemgrößen graphisch dar. Geben Sie die MFLOPS für die Einzel-Prozessorverarbeitung an!
- Nutzen Sie für das Verteilen und Einsammeln der Daten `MPI_Scatter()` und `MPI_Gather()` und führen Sie die Messungen noch einmal durch!
- Vergleichen Sie die beiden generellen Methoden der Aufgabenlösung unter 4.1 und 4.3 und 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 Version 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

Beim hier gestellten Problem, eine Vielzahl an Skalarprodukten parallel zu berechnen, ist zunächst festzustellen, dass jedes Punktprodukt für sich von allen anderen unabhängig berechnet werden kann. Darüber hinaus können partielle Punktprodukte, bei denen nur ein Bruchteil einer Zeile mit dem Vektor multipliziert wurden, einfach mittels Addition der Teilergebnisse aggregiert werden, solange alle Teilergebnis von disjunkten Abschnitten der Zeile stammen und diese in Gänze abdecken.
Mittels der MPI-Funktionalitäten für abgeleitete Datentypen lassen sich Speicherbereiche wie die hier verwendete Matrix $A \in \mathbb{R}^{m \times n}$ in flexible Raster zerlegen und übertragen. `MPI_Type_vector` erstellt einen Datentyp, der zusammenhängende Blöcke von Daten mit konstantem Abstand zueinander repräsentiert[2]. So lassen sich etwa jede k-te Zeile (Blocklänge m, Stride $2 \cdot m$) oder jedes p-te Element (Blocklänge 1, Stride p) der Matrix adressieren.
`MPI_SCATTER` stellt praktisch eine verkürzte Schreibweise eines Loops aus `MPI_Send` und `MPI_Recv` dar, ist jedoch insofern eingeschränkt, dass jeder Prozess die selbe Menge an Daten empfangen muss[3].

4 Implementierung

Das geschriebene C-Programm befindet sich im Anhang als Listing 1. Für die Übermittlung der Subarrays wurde verschiedene MPI-Vektor-Konfigurationen erprobt, wie etwa hier:

```
MPI_Datatype p_stride_t;
MPI_Type_vector((m*n)/p, 1, p, MPI_DOUBLE, &p_stride_t);
MPI_Type_commit(&p_stride_t);
```

Dabei erhielten alle p Prozesse jeweils versetzt jedes p-te Element der Matrix.
Dabei erwiesen sich komplexere Verteilungsschema jedoch zum Einen als komplex in der korrekten Implementation, da die Ermittlung der zu bearbeitenden Indizes in der eigentlichen Multiplikations-Routine entsprechend flexibel sein muss. Zum Anderen waren alle Varianten, bei denen der einzelne Prozess schließlich auf im Speicher weiträumig verteilte Daten zugreifen muss, als deutlich nachteilig hinsichtlich der Performance, da die aufs Lokalitätsprinzip ausgelegten Caching-Mechanismen nicht greifen können. In Folge dessen kristallisierte sich folgende Konfiguration als optimal heraus, bei der schlicht die Matrix in p aufeinanderfolgende Chunks aufgeteilt wird:

```
MPI_Datatype chunk_t;
MPI_Type_vector(1, (m*n)/p, (m*n), MPI_DOUBLE, &chunk_t);
MPI_Type_commit(&chunk_t);
```

Diese ist jedoch auch so simpel durch Pointer-Arithmetik darstellbar, dass sie in der endgültigen Programmversion durch solche ersetzt wurde. Hier wäre auch eine deutliche Optimierung dahingehend möglich, dass für die Verarbeitung der Subarrays nicht der selbe Speicherplatz wie für die gesamte Matrix allokiert werden müsste, was sich bei Verwendung der MPI-Vektoren nicht vermeiden ließe.
Die eigentliche Zeilen-Vektor-Multiplikation lässt sich dann für den Prozess mit Rank `world_rank` leicht darstellen:

```
int block_len = (m*n) / p;
for (int i = world_rank * block_len; i < block_len * (world_rank + 1); i++) {
    row = i / n;
    col = i \% n;
    result_local[row] += b[col] * A[i];
}
```

Weiterhin wurde dafür Sorge getragen, dass das Programm auch für solche Ausdehnungen der Matrix korrekt funktioniert, die kein ganzes Vielfaches der Prozessorzahl ergeben. In diesem Fall berechnet der root-Rank die Produkte für den verbleibenden „Rest“:

```
if (world_rank == 0) {
    for (int i = N - 1; i > block_len * p - 1; i--) {
        row = i / n;
```

```
4           col = i \% n;
5           result_local[row] += b[col] * A[i];
6       }
7   }
```

Wie eingangs beschrieben, ist die Aggregation von disjunkten Teilergebnissen beim hier gestellten Problem durch simple Addition darstellbar. Hierfür wurde die bereits bekannte MPI-Routine `MPI_Reduce` genutzt, weshalb die Teilergebnisse nicht explizit manuell zum Master-Prozess zurück transferiert und aggregiert werden müssen.
Die Zeitmessung erfolgt auf dem Root-rank mittels `clock_gettime(CLOCK_MONOTONIC, ...)` und erfasst dabei nur den Codeabschnitt für die eigentliche Übertragung der Daten, die Multiplikation sowie die Aggregation des Ergebnisses.

4.1 Bedienungshinweise

Das Programm erwartet vom Nutzer die Übergabe dreier Argumente: m, n sowie ob die Scatter-Funktionalität von MPI genutzt werden soll (1) oder nicht (0).
Anschließend generiert das Programm 100 mal eine zufällige Matrix und einen Vektor, und führt die parallelisierte zeilenweise Multiplikation durch. Die gemessene Zeit für jeden Durchgang, sowie der erreichte Gesamt-Durchsatz in MFLOPS werden in eine Datei mit dem Namen „kppr_4.out“ als CSV geschrieben.
Das SBATCH-Skript zur Durchführung der Messungen auf dem Taurus-Cluster befindet sich als Listing 2 im Anhang.

5 Ergebnisse

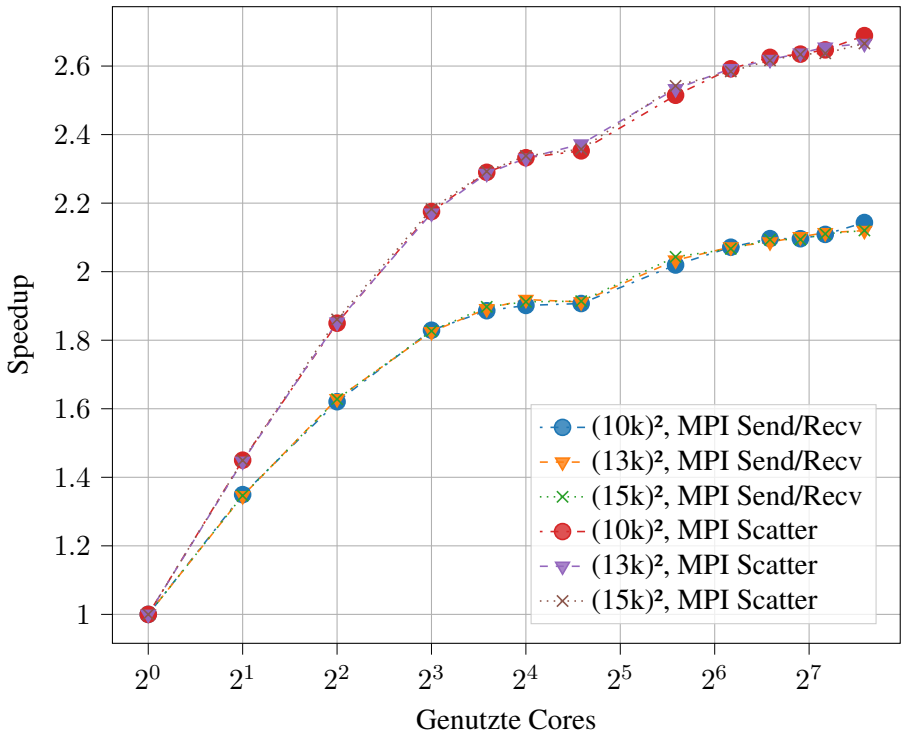

Abbildung 1: Speedups für drei verschiedene Problemgrößen und die zwei Übermittlungstechniken gegenüber der eingesetzten Anzahl an Prozessoren. Gemessen wurden die Laufzeiten für jeweils Problemlösungen Systeme und darüber gemittelt.

Für die Messungen wurden die Problemgrößen 10000×10000, 13000×13000 und 15000×15000 gewählt, um bei praktikablem Messaufwand und innerhalb der Grenzen des vorhandenen Arbeitsspeichers ein Bild vom generellen Skalierungsverhalten zu erhalten. Für fast alle Sets von 100 Messungen lagen die Standardabweichungen gegenüber dem Mittelwert bei unter einem Prozent, weshalb im Weiteren nur die gemittelten Werte angegeben werden.
Die resultierenden Speedups für alle Messreihen sind der Vergleichbarkeit halber zusammen in Abbildung 1 dargestellt. Die Kurven für die verschiedenen Problemgrößen überlagern sich dabei in allen Fällen sehr stark, abgesehen von den absoluten Laufzeiten weisen diese also alle das selbe grundlegende Verhalten auf. Die Speedups fallen durchgehend mäßig aus, es ist zu beachten, dass lediglich die Achse der genutzten Cores einen Log-Scale hat. Selbst unter Einsatz hunderter Cores lässt sich nur ein Speedup unter 3 erreichen. Das grundlegende Problem dürfte hier sein, dass die Berechnung des Punktprodukts sehr wenige, nämlich genau zwei, Fließkommaoperationen pro Datum in der Matrix benötigt. Jeglicher Datentransfer nimmt damit einen erheblichen Anteil an der Laufzeit ein, da Datenzugriffe in aller Regel deutlich höhere Latenz mit sich bringen als einzelne arithmetische Operationen benötigen. Die Verteilung der Subarrays ist darüber hinaus durch den Root-Rank als Flaschenhals limitiert.
Unter diesen Gesichtspunkten passt es ins Bild, dass die MPI-Scatter-Routine eine bessere Performance bietet, als die sequentiell blockierende Übertragung durch aufeinanderfolgende MPI-Send/MPI-Recv Anweisungen. Die mit MPI Scatter erfolgten Messungen zeigen durchgehend höhere Speedups, im Rahmen der am Root-Rank zur Verfügung stehenden Bandbreite könnte die MPI-Implementierung hier die Übertragung parallelisieren.
Die Varianten, die MPI Scatter nutzen, zeigen durchgehend deutlich höhere Speedups. Dabei ist aber zu beachten, dass sie insbesondere für Prozessorenzahlen unter 24 deutlich höhere Laufzeiten aufweisen. Der Vorsprung im Speedup stellt insofern nur ein Aufholen dar, im Bereich von 48 Kernen und mehr gleichen sich die Laufzeiten praktisch vollständig an.

Beide Übertragungstechniken zeigen einen ähnlichen Kurvenverlauf, insbesondere fällt der Knick beim Überschreiten von 24 Cores (einer vollständigen physischen Node) ins Auge. Dieser legt nahe, dass die Speicherbandbreite der zwei Sockets innerhalb der einen Node zunächst schon bei der Nutzung von 12 Cores fast vollständig ausgelastet war, die Hinzunahme weiterer phyischen Nodes bringt dagegen noch einmal Potential für weiteren Speedup, da damit zusätzlicher Speicher zur Verfügung steht, auf den unabhängig von den bisherigen 24 Prozessen zugegriffen werden kann. Dies ist also ein weiteres Indiz für die Speicherbandbreiten-limitierte Natur des hier betrachteten Problems.
Der arithmetische Durchsatz bei Einzelprozessorverarbeitung beträgt für MPI Send/Recv über alle Problemgrößen hinweg 494 MFLOPS, für MPI Scatter 392 MFLOPS. In allen Messreihen ist die Standardabweichung mit unter $0,2$ MFLOPS sehr gering.

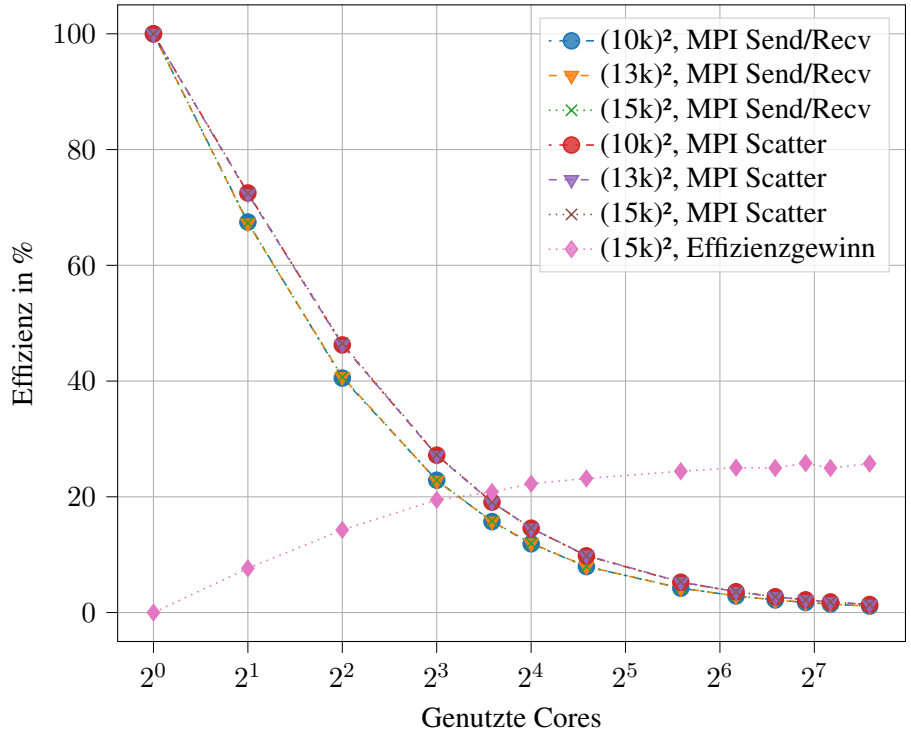

Abbildung 2: Effizienz der Parallelisierung als Anteil der sequentiellen Laufzeit an der tatsächlich genutzten CPU-Zeit für drei verschiedene Problemgrößen und die zwei Übermittlungstechniken gegenüber der eingesetzten Anzahl an Prozessoren. Zusätzlich ist der relative Effizienzgewinn von MPI Scatter gegenüber MPI Send/Recv dargestellt. Gemessen wurden jeweils die Laufzeiten für 100 Problemlösungen und darüber gemittelt.

In Abbildung 2 sind die erreichten Effizienzen dargestellt. Entsprechend der niedrigen Speedups gehen diese sehr schnell gegen Null, wobei die MPI-Scatter-Varianten etwas höhere Werte halten können. Der relative Unterschied zwischen den Übertragungstechniken ist auch noch einmal gesondert als Effizienzgewinn dargestellt. Die Kurven verlaufen für hohe Zahlen an genutzten Cores also nicht so ähnlich wie es optisch naheliegt, sondern MPI Scatter liegt relativ konstant gut 20% über MPI Send/Recv.

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] *RookieHPC*. https://www.rookiehpc.com/mpi/docs/mpi_type_vector.php, 2020. – [Online; Stand 19. März 2021]

[3] OPEN MPI PROJECT: *OpenMPI Documentation*. https://www.open-mpi.org/doc/current/man3/MPI_Scatter.3.php, 2020. – [Online; Stand 19. März 2021]

A Programmcode

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

#define REPS 100

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

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

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

    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    int scatter = atoi(argv[3]);
    int N = m*n;

    FILE *outf;
    if (world_rank == 0){
    outf = fopen("kppr_4.out", "w");
    }

    double *A;
    A = malloc(N*sizeof(double));
    double *b;
    b = malloc(n*sizeof(double));
    srandom(time(NULL));

    for(int rep = 0; rep < REPS+1; rep++) {
        if (world_rank == 0) {
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    A[i * n + j] = (random() % 20) - 10;
                }
            }

            for (int j = 0; j < n; j++) {
                b[j] = (random() % 20) - 10;;
            }
        }

        struct timespec tStart, tEnd;
        clock_gettime(CLOCK_MONOTONIC, &tStart);

        MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        int block_len = N / p;

        if (scatter) {
            MPI_Scatter(A, block_len, MPI_DOUBLE, &A[block_len * world_rank], block_len,
                        MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
60          } else {
61              for (int i = 1; i < p; i++) {
62                  if (world_rank == 0) {
63                      MPI_Send(&A[i * block_len], block_len, MPI_DOUBLE, i, 0,
    MPI_COMM_WORLD);
64                  }
65                  if (world_rank == i) {
66                      MPI_Recv(&A[i * block_len], block_len, MPI_DOUBLE, 0, 0,
67                               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
68                  }
69              }
70          }
71
72          double result_local[m];
73          for (int i = 0; i < m; i++) {
74              result_local[i] = 0;
75          }
76
77
78          int row, col;
79
80          for (int i = world_rank * block_len; i < block_len * (world_rank + 1); i++)
    {
81              row = i / n;
82              col = i % n;
83              result_local[row] += b[col] * A[i];
84          }
85
86          if (world_rank == 0) {
87              for (int i = N - 1; i > block_len * p - 1; i--) {
88                  row = i / n;
89                  col = i % n;
90                  result_local[row] += b[col] * A[i];
91              }
92          }
93
94          double result[m];
95          MPI_Reduce(result_local, result, m, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
96
97          clock_gettime(CLOCK_MONOTONIC, &tEnd);
98          double timeTaken = tEnd.tv_sec - tStart.tv_sec + (tEnd.tv_nsec - tStart.
    tv_nsec) * 1e-9; // in seconds
99
100         if (world_rank == 0 && rep > 0) {
101             fprintf(outf, "%.9f, %.3f\n", timeTaken, (2 * N) / (1000 * 1000 *
    timeTaken));
102         }
103     }
104     MPI_Finalize();
105 }
```

Listing 1: Der verwendete Programmcode für die durchgeführten Messungen.

B SBATCH-Skript

```
#!/bin/bash

#SBATCH --nodes 8

#SBATCH --ntasks-per-node 24

#SBATCH --cpus-per-task 1

#SBATCH --mem-per-cpu=2000

#SBATCH --partition=haswell

#SBATCH --exclusive

# jobname:
#SBATCH -J kppr_4
#SBATCH --output %x.%j.log

# Load required modules
module load CMake/3.11.4-GCCcore-7.3.0
module load OpenMPI

cd /home/s4690420/kppr/task4

for cores in 192 144 120 96 72 48 24 16 12 8 4 2 1
do
  for scatter in 1 0
  do
    for m in 10000 13000 15000
    do
      srun --ntasks=${cores} --nodes=$(((cores+23)/24)) --cpus-per-task=1 ./kppr_4 $((m)) $((m)) ${scatter}
      mv kppr_4.out "results/${cores}_${scatter}_$((m)).csv"
    done
  done
done
```

Listing 2: Das verwendete SBATCH-Skript um die Messungen auf dem Taurus-Cluster durchzuführen.