# **Protokoll 2: Verteiltes Rechnen mittels MPI**

**Gruppe 9: Marcel Beyer, Martin Carnein, Franz Lehmann**

## Aufgabe 1:

Berechnung der Peak Performance mit folgender Formel:

Laut Intel Webseite besitzt der *Intel Xeon Platinum 8470 Prozessor* zwei AVX-512 FMA-Einheiten, sodass sich bei Double Precision (64 bit) daraus 32 Instruktionen pro Zyklus ergeben. Die Basisfrequenz pro Kern ist 2.00 GHz.

Damit ergibts sich folgende Lösung:

## Aufgabe 2:

**MPI - (Message Passing Interface):**

Ist ein Interface für Programmierung mit parallelen Prozessen und der Standard für parallele Berechnungen in Hochleistungsrechnern. Es kann in mehreren Programmiersprachen genutzt werden und hat verschiedene Implementierungen (MPICH, OpenMPI, Vendor). MPI funktioniert nach SPMD-Modell: Viele Prozesse führen den gleichen Programmcode aus.

**Nützliche Funktionen für das Berechnen einer Matrix-Multiplikation:**

MPI\_Init():

* notwendig um die MPI-Umgebung zu initialisieren

MPI\_Comm\_rank():

* ermittelt für jeden Prozess die eigene Prozessnummer
* damit kann unterschiedliches Verhalten der Prozesse implementiert werden

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size):

* durch rank und size lassen sich die Aufgaben verschieden skalieren und verteilen z.B. um load balancing zu optimieren

MPI\_Recv(), MPI\_Send():

* sind blockierende, asynchrone Operationen, um Nachrichtentransfer zwischen Prozessen zu realisieren
* alternativ kann auch nicht blockierende Version genutzt werden

MPI\_Barrier():

* wird genutzt, um darauf zu warten, dass alle Prozesse ihre Berechnungen und Nachrichtentransfers abgeschlossen haben, bevor die Matrixmultiplikation abgeschlossen wird.

## Aufgabe 3:

Folgende MPI Versionen sind verfügbar:

* OpenMPI:
  + Versions:
    - OpenMPI/4.1.1
    - OpenMPI/4.1.4
* impi:
  + Versions:
    - impi/2021.6.0
    - impi/2021.7.1

## Aufgabe 4:

Den verwendeten Compiler sowie die verwendete MPI Version kann wie folgt ermittelt werden:

**Kompilieren:**

OpenMPI: $ mpicc my\_mpi\_application.c -o my\_mpi\_application

IMPI: $ mpiicc my\_mpi\_application.c -o my\_mpi\_application

**Compiler finden:**

$ mpirun --version

**Version finden:**

OpenMPI: $ mpicc -v

IMPI: $ mpiicc -v

## Aufgabe 5:

Slurm ist ein batch - System, das Ressourcen verwaltet und Jobs scheduled. Will man auf dem HPC-System einen Job ausführen, spezifiziert man die benötigten Ressourcen, übergibt diese Slurm, welches dann den Job scheduled.

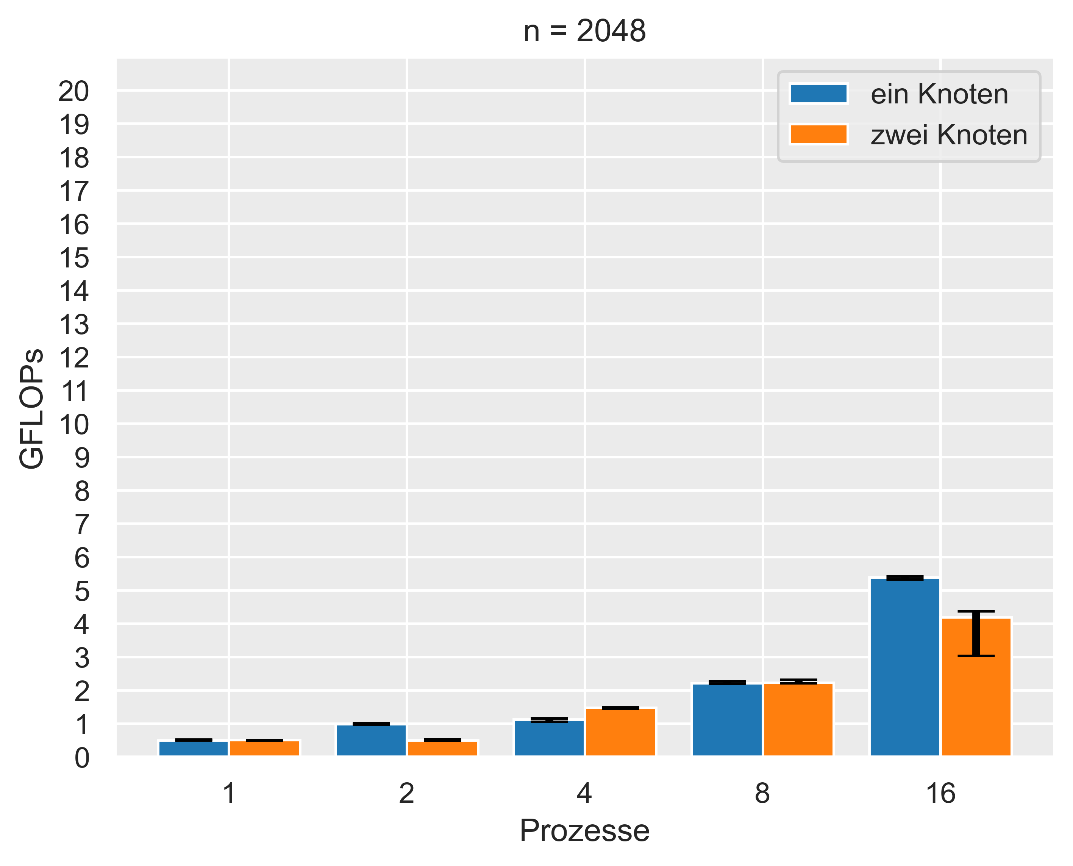
Folgende Config für srun muss genutzt werden um einen job mit 20 Prozessen gleichmäßig verteilt auf 2 Knoten zu starten:

*srun -n 20 -N 2 -t <time> <job\_name>*

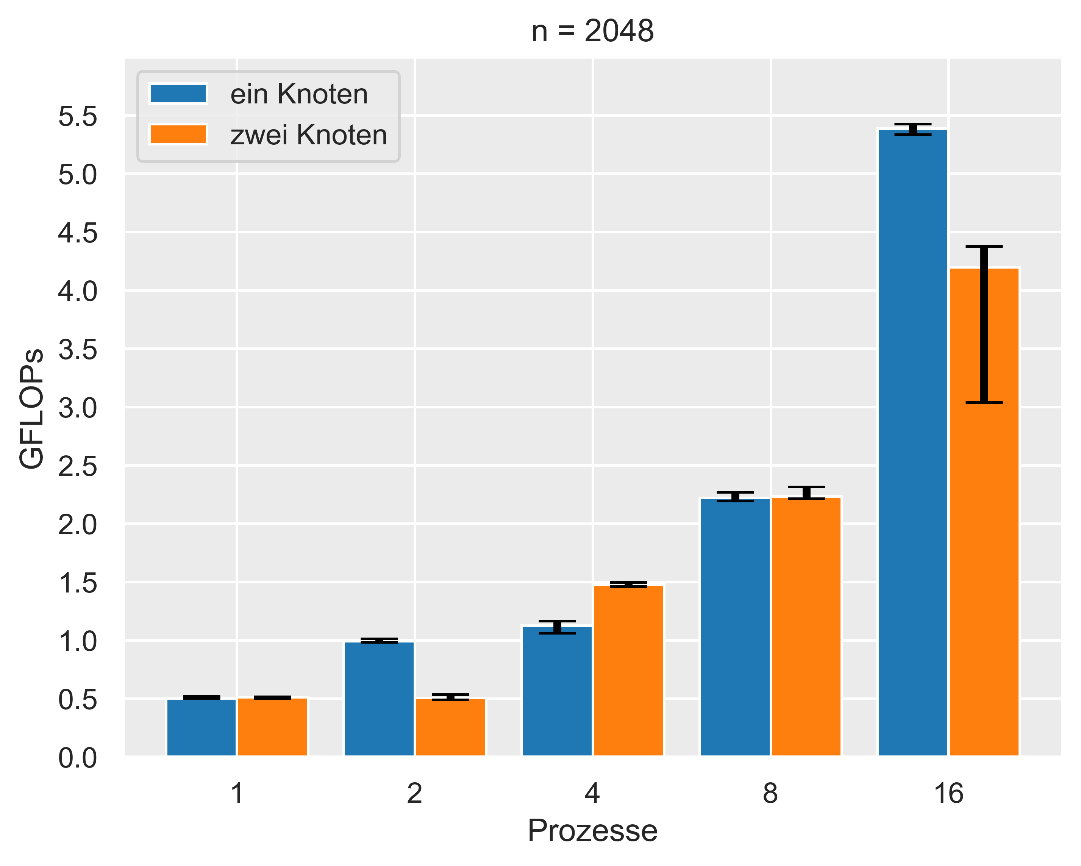
## Aufgabe 6:

In folgenden Diagrammen wurde als Mittelwert der Median gewählt.

Für alle Diagramme wurde auf der y-Achse ein Maximum von 20 GFLOPs gewählt, damit man die Diagramme einfach vergleichen und Unterschiede direkt sehen kann.



Da bei einer Matrixgröße von n=2048 die Messwerte nicht so hoch waren, haben wir uns dazu entschieden hier noch ein Diagramm bereit zu stellen, wo die Werte genauer erkennbar sind:



## zu Aufgabe 6:

|  |  |
| --- | --- |
|  |  |
|  | |

In den Diagrammen kann man gut erkennen, dass mit erhöhter Matrixgröße auch die Leistung abnimmt.

## zu Aufgabe 6:

Quellcode für die Matrixmultiplikation mit MPI:

void mpi\_mat\_mul(const float\* mat1, const float\* mat2, int n, int start\_mat\_index, int end\_mat\_index, float\* mat\_out) {  
 int start\_i = floor(start\_mat\_index / n);  
 int start\_j = start\_mat\_index % n;  
 int end\_i = floor(end\_mat\_index / n);  
 int end\_j = end\_mat\_index % n;  
 int out\_index = 0;  
  
 if (start\_i == end\_i) {  
 // first row  
 for(int j = start\_j; j <= end\_j; j++) {  
 mat\_out[out\_index] = mat1[start\_i\*n] \* mat2[j];  
 for (int k = 1; k < n; k++) {  
 mat\_out[out\_index] += mat1[start\_i\*n+k] \* mat2[k\*n+j];  
 }  
 out\_index++;  
 }  
 } else {  
 // first row  
 for (int j = start\_j; j < n; j++) {  
 mat\_out[out\_index] = mat1[start\_i \* n] \* mat2[j];  
 for (int k = 1; k < n; k++) {  
 mat\_out[out\_index] += mat1[start\_i \* n + k] \* mat2[k \* n + j];  
 }  
 out\_index++;  
 }  
  
 for (int i = start\_i + 1; i < end\_i; i++) {  
 for (int j = 0; j < n; j++) {  
 mat\_out[out\_index] = mat1[i \* n] \* mat2[j];  
 for (int k = 1; k < n; k++) {  
 mat\_out[out\_index] += mat1[i \* n + k] \* mat2[k \* n + j];  
 }  
 out\_index++;  
 }  
 }  
  
 // last row  
 for (int j = 0; j <= end\_j; j++) {  
 mat\_out[out\_index] = mat1[end\_i \* n] \* mat2[j];  
 for (int k = 1; k < n; k++) {  
 mat\_out[out\_index] += mat1[end\_i \* n + k] \* mat2[k \* n + j];  
 }  
 out\_index++;  
 }  
 }  
}

## zu Aufgabe 6:

int compare( const void\* a, const void\* b)  
{  
 long double int\_a = \* ( (long double\*) a );  
 long double int\_b = \* ( (long double\*) b );  
  
 if ( int\_a == int\_b ) return 0;  
 else if ( int\_a < int\_b ) return -1;  
 else return 1;  
}  
  
  
int main(int argc, char\* argv[]) {  
 if (argc != 2) {  
 printf("Usage: %s <function number> <matrix size>\n", argv[0]);  
 return 1;  
 }  
 if (atoi(argv[1]) < 0 || atoi(argv[1])%2 != 0){  
 printf("Wrong Matrix size <n> needs to fulfill: 2^x = n !\n");  
 return 1;  
 }  
  
 int matrix\_size;  
 //printf("argv[1]: %s\n", argv[1]);  
 matrix\_size = atoi(argv[1]);  
  
 int n = matrix\_size;  
 int evals = 10;  
 struct timeval start, end;  
 long double measures[10] = {0};  
 long double avg\_time = 0;  
  
 float\* mat1 = (float\*) malloc(n\*n\*sizeof(float));  
 float\* mat2 = (float\*) malloc(n\*n\*sizeof(float));  
 for (int i = 0; i < n; i++) {  
 for(int j = 0; j < n; j++) {  
 mat1[i\*n+j] = (float)(i+j);  
 mat2[i\*n+j] = (float)(i+j);  
 }  
 }  
  
 int rank, size;  
  
 // MPI section  
 MPI\_Init(NULL, NULL);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
  
 for (int e = 0; e < evals; e++) {  
 // start time measurement on rank 0  
 if (rank == 0) {  
 gettimeofday(&start, NULL);  
 }  
  
 int start\_mat\_index = floor(n \* n / size \* rank);  
 int end\_mat\_index;  
 if (rank == size - 1) {  
 end\_mat\_index = n \* n - 1;  
 } else {  
 end\_mat\_index = floor(n \* n / size \* (rank + 1)) - 1;  
 }  
 int sequence\_length = end\_mat\_index - start\_mat\_index + 1;  
  
 float \*mat\_out;  
 if (rank == 0) {  
 // printf("start time: %ld\n", start.tv\_sec \* 1000 \* 1000 + start.tv\_usec);  
 mat\_out = malloc(n \* n \* sizeof(float));  
 } else {  
 mat\_out = malloc(sequence\_length \* sizeof(float));  
 }  
  
 mpi\_mat\_mul(mat1, mat2, n, start\_mat\_index, end\_mat\_index, mat\_out);  
  
 if (rank == 0) {  
 // printf("rank %i hat fertig gerechnet\n", rank);  
 for (int i = 1; i < size; i++) {  
 int start\_for\_rank = floor(n \* n / size \* i);  
 int end\_for\_rank;  
 if (i == size - 1) {  
 end\_for\_rank = n \* n - 1;  
 } else {  
 end\_for\_rank = floor(n \* n / size \* (i + 1)) - 1;  
 }  
 int sequence\_length\_for\_rank = end\_for\_rank - start\_for\_rank + 1;  
 MPI\_Recv(&mat\_out[start\_for\_rank], sequence\_length\_for\_rank, MPI\_FLOAT, i, 0, MPI\_COMM\_WORLD,  
 MPI\_STATUS\_IGNORE);  
 }  
 gettimeofday(&end, NULL);  
 // printf("end time: %ld\n", end.tv\_sec \* 1000 \* 1000 + end.tv\_usec);  
 long long microseconds = ((end.tv\_sec - start.tv\_sec) \* 1000 \* 1000 + end.tv\_usec - start.tv\_usec);  
 // printf("time: %lld\n", microseconds);  
 avg\_time += (long double)(microseconds)/evals;  
 long long tmp = (long long)n;  
 long long no\_fops = tmp \* tmp \* (2 \* tmp - 1);  
 measures[e] = (long double) no\_fops / microseconds;  
 } else {  
 MPI\_Send(mat\_out, sequence\_length, MPI\_FLOAT, 0, 0, MPI\_COMM\_WORLD);  
 }  
 free(mat\_out);  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 }  
 MPI\_Finalize();  
 free(mat1);  
 free(mat2);  
 return 0;  
}