Protokoll 2: Verteiltes Rechnen mittels MPI

Gruppe 9: Marcel Beyer, Martin Carnein, Franz Lehmann

Aufgabe 1:

Berechnung der Peak Performance mit folgender Formel:

$$FLOPS = \frac{Instructions}{Cycle} * \frac{Clock\ Rate}{Core} * NumberOfCores$$

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:

$$FLOPS = 32\frac{Instructions}{Cycle} * \frac{2.00\ GHz}{Core} * 52\ Cores = 3{,}328\frac{TFLOP}{s}$$

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

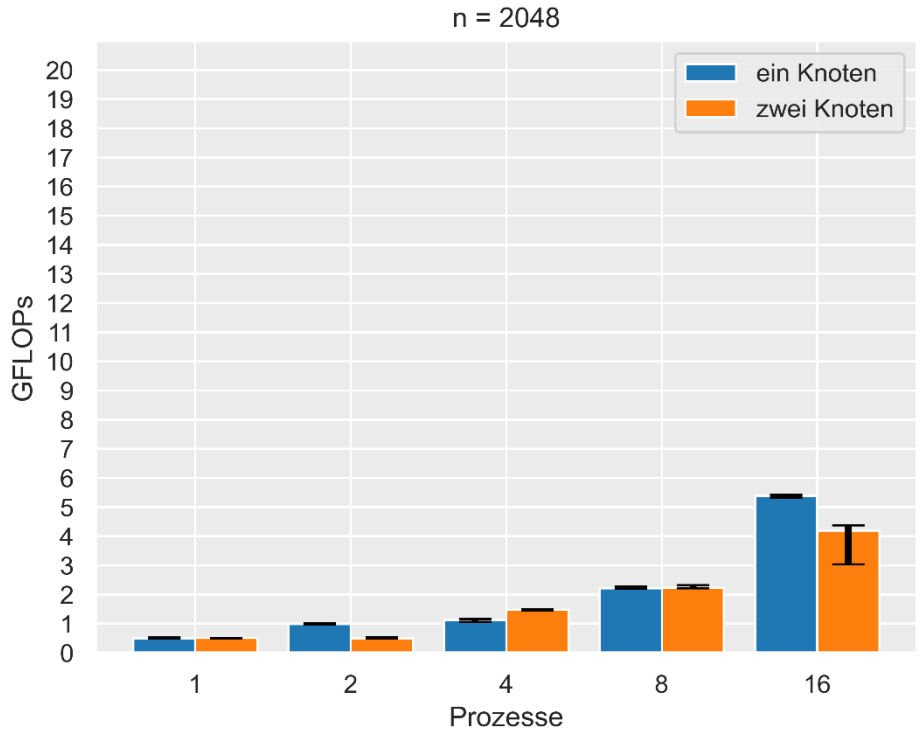
Aufgabe 5:

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.

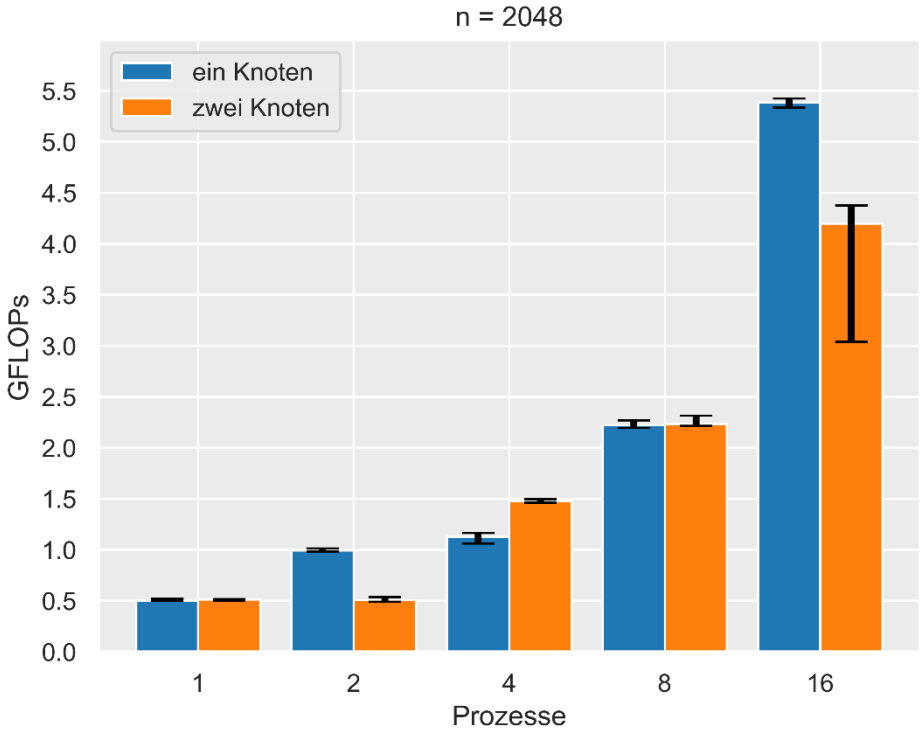
Folgende Config für srun muss genutzt werden um einen job mit 20 Prozessen gleichmäßig verteilt auf 2 Knoten zu starten:
srun -n 20 -N 2 -t <time> <job_name>

Aufgabe 6:

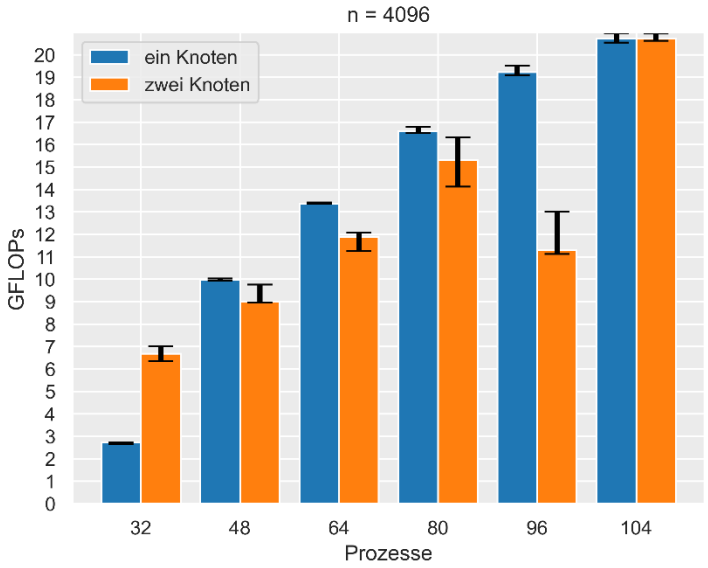
In folgenden Diagrammen wurde als Mittelwert der Median gewählt.
Für alle Diagramme wurde auf der y-Achse ein Maximum von 20 GFLOPs gewählt, damit man die Diagramme einfach vergleichen und Unterschiede direkt sehen kann.

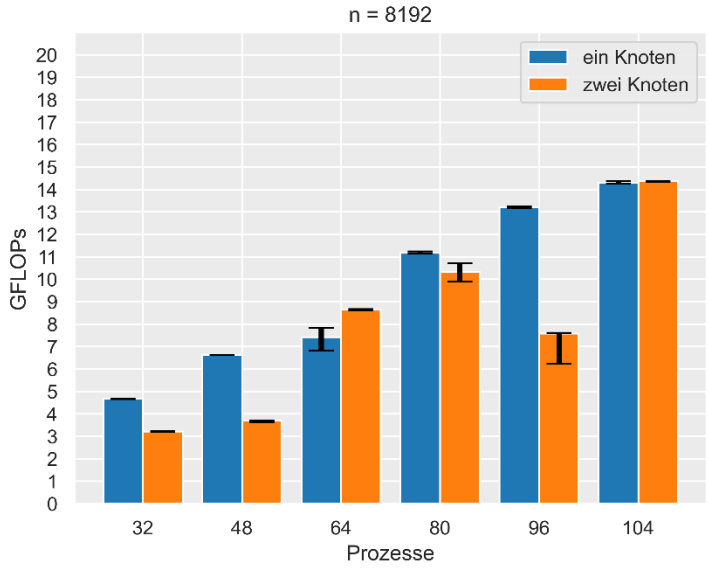


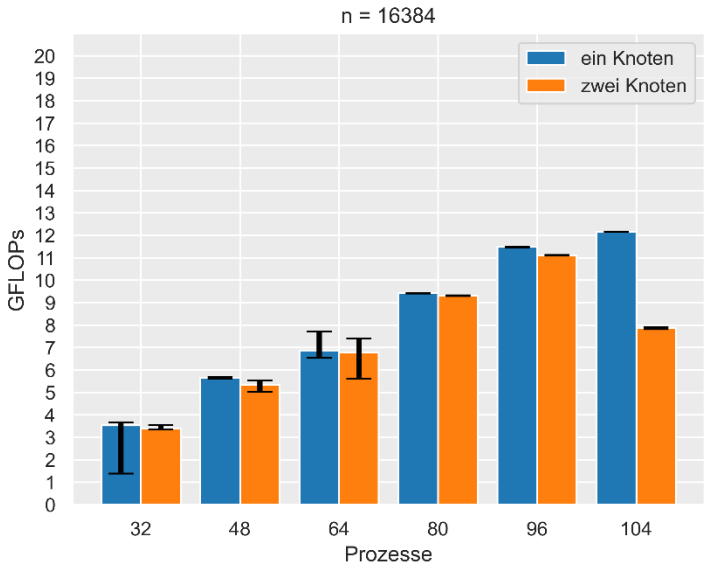
Da bei einer Matrixgröße von n=2048 die Messwerte nicht so hoch waren, haben wir uns dazu entschieden hier noch ein Diagramm bereit zu stellen, wo die Werte genauer erkennbar sind:



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;
}
```