Einführung in die technische Informatik

Rechnerarchitektur Praktikum 2: verteilte Matrixmultiplikation

Praktikumsbericht Gruppe 1: Benjamin Probst, Tim Hanel

Aufgabe 1:

> Kernanzahl × Taktfrequenz in GHz × CPU-Instruktionen pro Takt = Rechenleistung in GigaFlops

Daher 12 Kerne * 2,5Ghz Taktfrequenz*2 Ausführungseinheiten (FMA) * 4 SIMD Operationen/ FMA Einheit * 2 Operationen / FMA-Einheit = 480 Gigaflop ohne boost Takt
Und 12 Kerne * 3,3Ghz Taktfrequenz*2 Ausführungseinheiten (FMA) * 4 SIMD Operationen/ FMA Einheit * 2 Operationen / FMA-Einheit = 633.6 Gigaflop mit boost Takt

Aufgabe 2:

Um eine Matrixmultiplikation mithilfe von MPI zu beschleunigen können folgende Funktionen helfen:

```
MPI_Init(&argc, &argv); //initialisiert die MPI Kommunikation zwischen allen Prozessen
MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster); //initialisiert die Cluster Größe
MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank); //initialisiert die Prozessnummer

MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm,
communicator); //kann Nachrichten an Zielprozess senden
MPI_Scatter(void* sendbuf, intsendcount, MPI_Datatypesendtype,
void* recvbuf, intrecvcount, MPI_Datatyperecvtype, introot, MPI_Commcomm //kann vom root
Prozess Daten sortiert nach Reihenfolge an anderen Prozessen gestaffelt senden

MPI_Recv(void* data, int count, MPI_Datatype datatype, int from, int tag, MPI_Comm comm,
MPI_Status* status); //kann Nachricht von anderem Prozess empfangen
MPI_Gather(void* sendbuf, intsendcount, MPI_Datatypesendtype, void* recvbuf,
intrecvcount, MPI_Datatyperecvtype, introot, MPI_Commcomm) //kann an den root Prozess Daten
sortiert nach Reihenfolge von anderen Prozessen empfangen
```

Funktionen wie MPI_Allgather oder MPI_Allreduce werden nicht benötigt, da für die Verarbeitung nicht jeder Prozess im Besitz aller Daten sein muss. Eine redundante Verarbeitung ist nicht gewünscht.

Eine weitere interessante Funktion ist:

```
MPI_Alltoall(void* sendbuf, intsendcount, MPI_Datatypesendtype, void* recvbuf,
intrecvcount, MPI_Datatyperecvtype, MPI_Commcomm) // Kann Vektoren verschiedener Probleme
an multiple Prozesse kaskadieren, kann dazu genutzt werden mehrere Zeilen einer
Multiplikation an verschiedene Prozesse aufzuteilen

MPI_Reduce(void* send_data,void* recv_data,int count,MPI_Datatype datatype,
    MPI_Op op,int root,MPI_Comm communicator)
wird verwendet um die Ergebnisse der einzelnen Prozesse zusammenzuführen.

int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )

wird verwendet um den einzelnen Prozessen die Werte der Eingabematrizen zu
schicken.
```

Aufgabe 3:

Gefundene Modulabhängigkeit: scorep/scs5/trunk-pgi-ompi-cuda9.1 unter modenv/scs5. Dazu gehören untergeordnete Module: Module GCCcore/12.2.0zlib/1.2.12-GCCcore-12.2.0binutils/2.39-GCCcore-12.2.0GCC/12.2.0.

Aufgabe 4:

Mit dem Befehl module spider -r mpi lässt sich die verfügbare Version auf Taurus finden.

Version: OpenMPI/4.1.1-intel-compilers-2021.2.0
Kompilieren mit mpicc:
mpicc -O3 -march=native -o task2 task2_.c

Aufgabe 5:

Zunächst mit salloc -t 02:00:00 -p haswell --nodes=2 --tasks-per-node=12 –mem-per-cpu=1024 Ressourcen sichern.

Danach um 12 Prozesse gleichermaßen auf 2 Knoten zu starten lautet :
srun --ntasks=12 --tasks-per-node=6 --exclusive ./task2 oder
srun --ntasks=12 --mincpus=6 --exclusive ./task2

Aufgabe 6:

Compilerflags: mpicc -O3 -march=native -o task2 task2_.c

Verwendete Matrixfunktion mit Schleifenvertauschen (andere Optimierungen aus Aufgabe 1 oder Kombinationen haben geringere Performance gezeigt) :

```
void matmuljki(const double *input1, const double *input2, double *output) {
   int MPIInit=rank*(SIZE/size);
   int MPIStepBound=(rank+1)*(SIZE/size);
   for (int j = 0; j < SIZE; j++)  {            //jki
      for (int k = MPIInit; k < MPIStepBound; k++) {
         for (int i = 0; i < SIZE; i++){
               output[j * SIZE + i] += input1[j * SIZE + k] * input2[k * SIZE + i];
         }
           }
    }

}
```

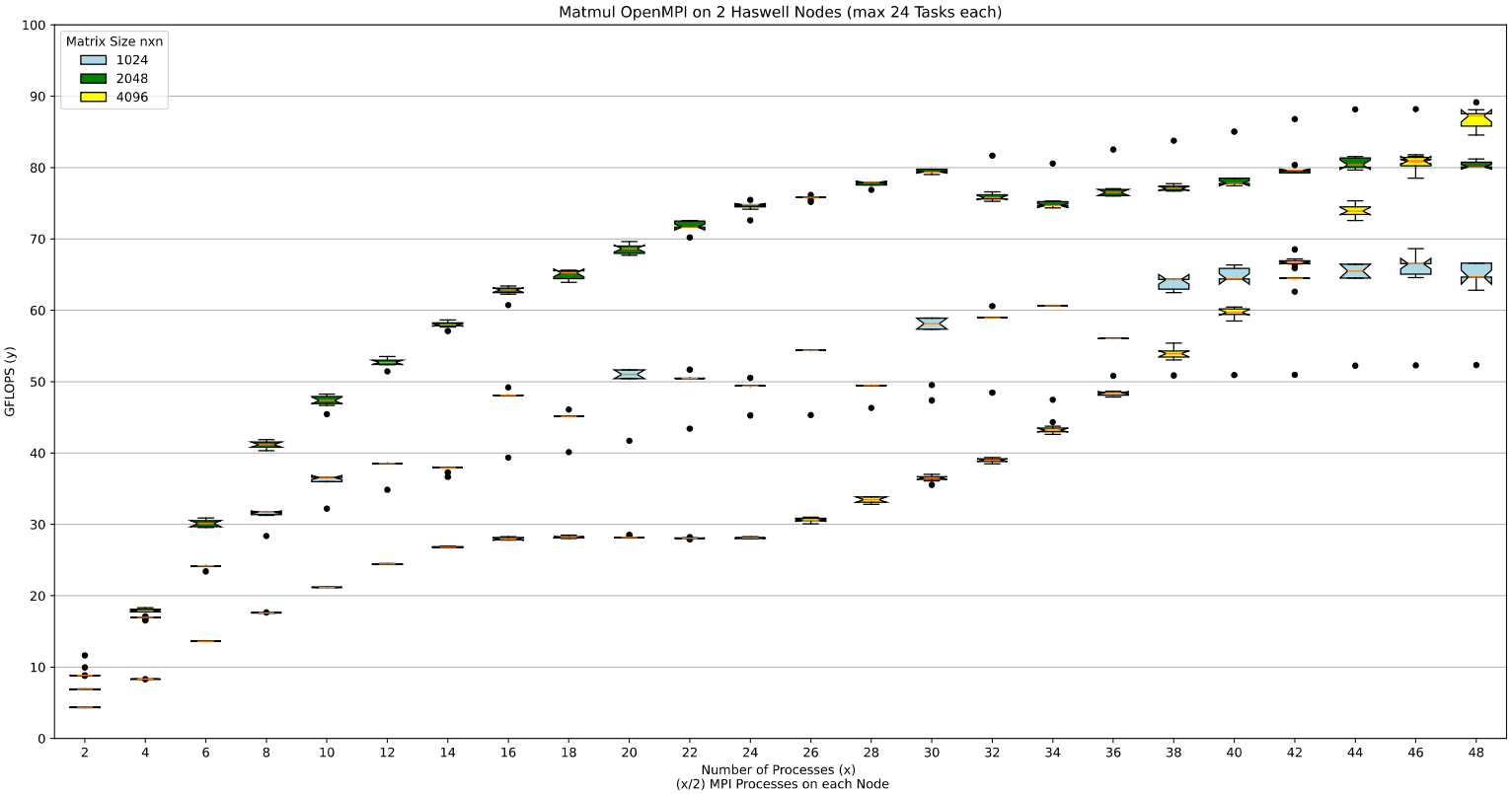
in der Main():

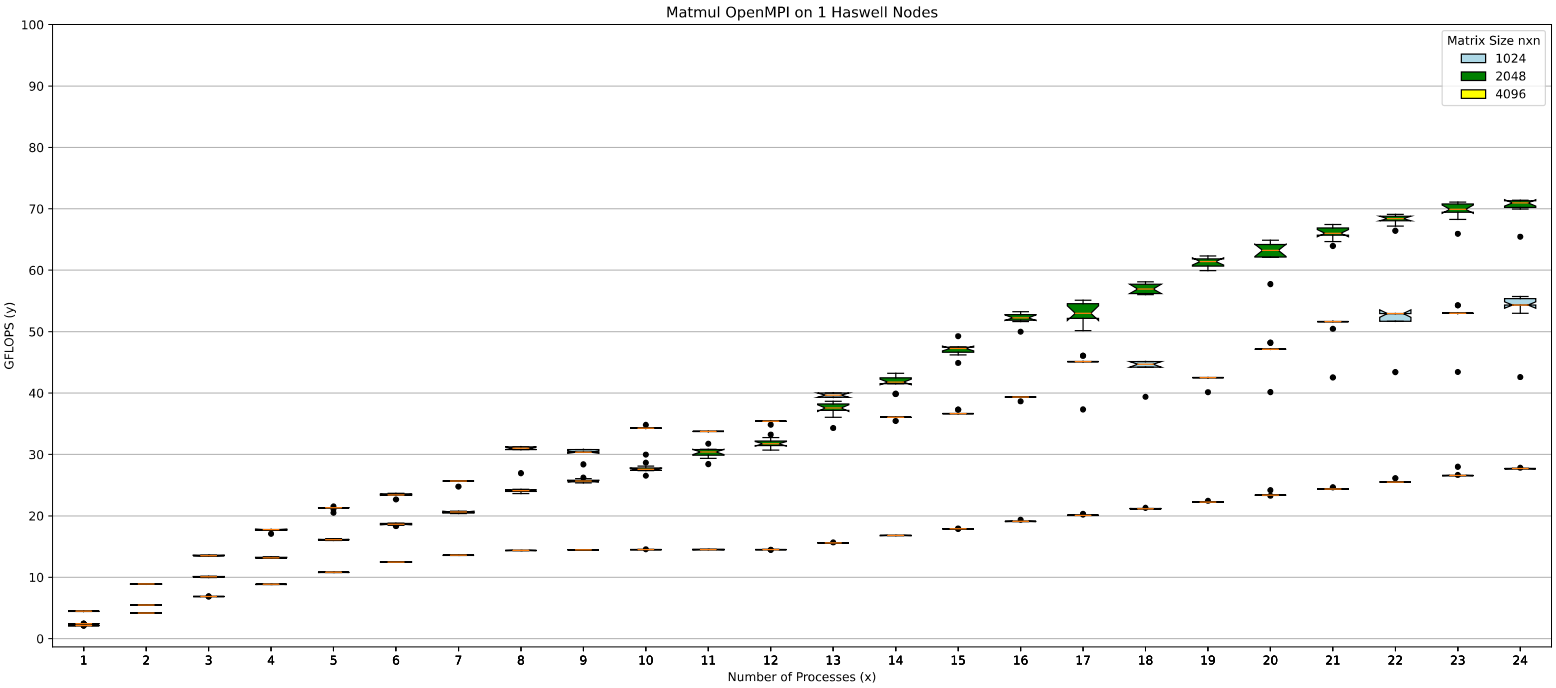
```
.
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
.
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(input1,SIZE*SIZE,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(input2,SIZE*SIZE,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0){
      gettimeofday(&time, NULL);
      millis = (time.tv_sec * (long long) 1000) + (time.tv_usec / 1000);
}
```

```
MPI_Barrier(MPI_COMM_WORLD);
matmuljki(input1,input2,output);
MPI_Reduce(output,finaloutput,SIZE*SIZE,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0){
        gettimeofday(&time, NULL);
```

Anmerkung: Die Verzögerung durch Bcast ist in unserem Fall nicht Teil der Zeitmessung, MPI_Reduce allerdings schon (MPI_Reduce führt im Gegensatz zu Bcast Berechnungen durch).

Die unten dargestellten Diagramme zeigen die Floating Point Performance für die Matrixmultiplikation bei Ausführung auf 1 bzw 2 Haswell Nodes (Intel(R) Xeon(R) E5-2680 v3 CPUs) auf dem Taurus HPC System und für die Matrixgrößen [1024,2048 und 4096] auf bis zu 48 Prozessen (maximal 24 pro Node). Zur Darstellung wurde ein Boxplot gewählt, welcher Minimum, Maximum und Median angibt. Da die Wertstreuung zum Teil sehr gering war sind diese Metriken für bestimmte Werte nicht zu unterscheiden. Ahmdals Law zeichnet sich in den meisten Kurven gut erkennbar ab.




Matmul OpenMPI on 1 Haswell Nodes
Matrix Size nxn
1024
2048
4096
GFLOPS (y)
Number of Processes (x)
0
10
20
30
40
50
60
70
80
90
100
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24