HPC Lab

Assignment 2

Erkin Kirdan - Manuel Rothenberg - Gabrielle Poerwawinata

1. Shared memory π-calculation

Method explanation

The value of π can be obtained from,

$$\pi = 4 \arctan(1)$$

because arctan(0) = 0 then π can be derived from the integration of function of ϕ(x) over [0,1], written as,

$$\int_0^1 \varphi(x) = \int_0^1 \frac{1}{1+x^2}$$

Using midpoint rule we can approximate definite integral with ,

$$\int_a^b \varphi(x)dx = \Delta x\ \varphi(x_1^*) + \Delta x\ \varphi(x_2^*) + .. + \Delta x\ \varphi(x_n^*)$$

$$= \Delta x\ [\varphi(x_1) + \varphi(x_2) + .. + \varphi(x_n)]$$

$$= \Delta x\ \sum_{i=1}^{n} \varphi(x_i)$$

Here Δx is length of subinterval,

$$\Delta x = h = \frac{1}{n}$$

Therefore, in the end value of π can be calculated as,

$$\pi = 4.h.\ \sum_{i=1}^{n} \varphi(x_i)$$

Calculating Speed Up Ratio

Speedup ratio is a ration of T_1 over T_N, where T_1 is the time needed to execute serial application, T_N is the time needed to execute the application with a certain number of processor.

$$S = T_1/T_N$$

Speedup for strong scaling

problem size	nthreads	speedup
100000	1	1
100000	5	0.9881289932
100000	10	0.9975573451
100000	15	1.035785052
100000	20	1.006391745
100000	25	1.011493808
100000	30	1.012434149
100000	35	1.015344573
100000	40	1.064947235
100000	45	1.012944231
100000	50	1.035620889

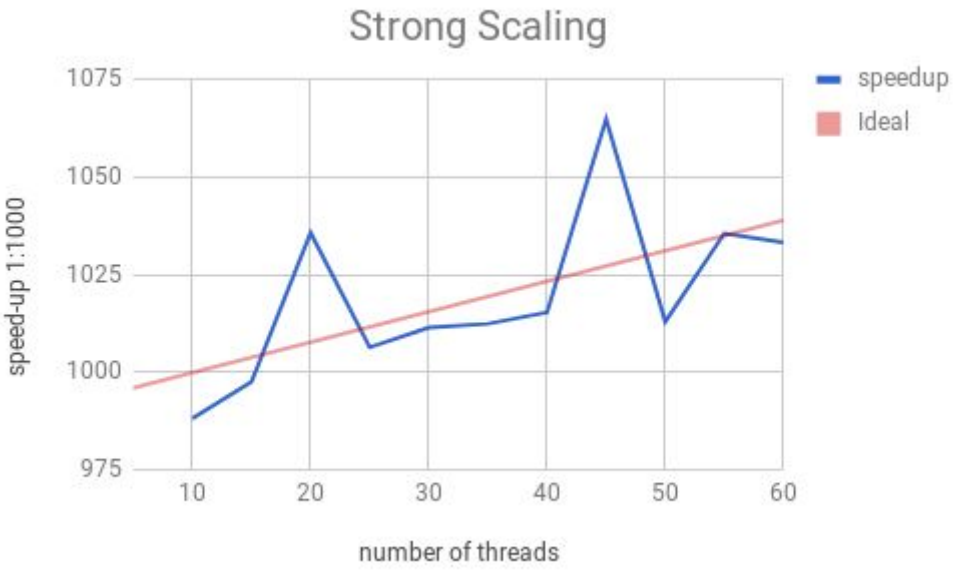
100000	55	1.033287211
100000	60	1.040402834
100000	64	1.054847042

Speedup for weak scaling

problem size	nthreads	speedup
100000	1	1
200000	5	0.9525840076
300000	10	0.938559322
400000	15	0.8683388704
500000	20	0.8320705463
600000	25	0.7959861128
700000	30	0.7696634177
800000	35	0.734247268
900000	40	0.6933259059
1000000	45	0.6757588293
2000000	50	0.4876942884
3000000	55	0.3784734792
4000000	60	0.3124006454
5000000	64	0.2657036261

Strong and weak scaling studies

In a strong scaling, the problem size of the program stays fixed but the number of processing elements are increased. By this we use problem size of 100000. In strong scaling a program is considered to scale linearly if the speed up is equal with the number processing used. As we can see, it is difficult to achieve linear speed up because of overhead increases together with the growth of number of processes used.



In weak scaling, the problem size assigned keep increasing along with the growth of threads number. Therefore linear scaling should be achieved if the run time stays constant while the number of workload and threads are doubled. However in this case, we are not getting the constant value of run time.

2. STREAM benchmark

Kernels of STREAM
STREAM has 4 compute kernels:

Name	Kernel	Bytes/Iteration	FLOPS/Iteration
Copy	a(i) = b(i)	16	0
Scale	a(i) = q*b(i)	16	1
Sum	a(i) = b(i) + c(i)	24	1
Triad	a(i) = b(i) + q*c(i)	24	2

- Copy benchmark measure the transfer rate. This should be one of the fastest memory operations, but also represents the common one like update operation and fetching values from memory.
- Scale benchmark adds a simple arithmetic operation to the copy benchmark. The operation fetches 2 values from memory a(i) and b(i) before writing it to a(i). The performance of this simple operation test can be used as an indicator of the performance of more complex operations.
- Sum benchmark adds a third operand. This benchmark is useful because the large pipelines that some processors possess. This benchmark fetches three rather than 2 memory. For larger arrays, this will quickly fill a processor pipeline, it useful to test the memory bandwidth filling the processor pipeline or the performance when the pipeline is full.
- Triad benchmark allows chained or overlapped or fused multiple add operations. It builds on the sum benchmark by adding an arithmetic operation to one of the fetched array values. Fused multiple add operations are important operation in many basic computations, such as dot products, matrix multiplication, polynomial evaluations.

Array size of STREAM
Array size of STREAM can be configured directly by the compiler. In this case we use array size of 72000000 to reach 16GiB

```
icc -qopenmp -DSTREAM_ARRAY_SIZE=72000000 stream.c -o
stream.omp.AVX2.72M.10x.icc
```

KNL Processor modes

- Cache mode : The MCDRAM is configured entirely as a last-level (L3) cache. All of the MCDRAM behave as a memory-side direct mapped cache in front of DDR4. As a result, there is only a single visible pool of memory and MCDRAM can be seen as high bandwidth L3 cache.
- Flat mode : The MCDRAM is used as a SW visible and Operating System (OS) managed addressable memory (as a separate NUMA node), so that memory can be selectively allocated to DDR4 or MCDRAM.

NUMA

- In quadrant mode the chip is divided into 4 virtual quadrants, but is exposed to the OS as a single NUMA domain. Each quadrants are spatially local to one group of memory controller. Memory request from a core will be forwarded to the tag directory which in turn is forwarded to the memory in case of a L2 miss. Since the tag directory and the memory channel are always in the same quadrant, the memory request will never need to go across quadrants, result in less mesh traffic and hence can provide the better performance. The division into quadrants is hidden from the OS and the memory appears to be one contiguous block from the user's perspective.

- Snc4/subnuma mode each quadrant or half of the chip is exposed as a separate NUMA domain to the OS. As a result there is a sense of localization and hence reduced mesh traffic congestion providing an opportunity for the absolute highest performance. In snc4 mode the chip is analogous to a 4 socket xeon therefore has potential for high performance. If in SNC4 mode cache traffic crosses NUMA boundaries, this path is more expensive than in the quadrant mode.

Result : quad,cache

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	16686.5	0.069160	0.069038	0.069494
Scale:	9355.4	0.123294	0.123137	0.123805
Add:	11185.8	0.154545	0.154482	0.154614
Triad:	10269.4	0.168501	0.168267	0.169052

Result: snc4,cache

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	16667.7	0.069156	0.069116	0.069190
Scale:	9483.1	0.121541	0.121479	0.121738
Add:	11366.4	0.152128	0.152027	0.152408
Triad:	10442.4	0.165634	0.165479	0.165824

Result quad, flat

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	15846.1	0.073203	0.072699	0.073738
Scale:	8666.5	0.133373	0.132925	0.133960
Add:	10927.6	0.158671	0.158132	0.159438
Triad:	9800.9	0.176889	0.176311	0.177625

Result snc4,flat

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	16222.2	0.071356	0.071014	0.072057
Scale:	8852.3	0.130847	0.130135	0.131791
Add:	11380.3	0.152307	0.151841	0.153525
Triad:	10056.4	0.172587	0.171831	0.173613

Memory Bandwidth Flat vs Cache modes

Flat mode requires modifications of the code or execution environment. Using HBM as addressable memory often requires more application development compared to HBM as cache mode. Application memory in flat mode is allocated to DDR4 by default so we need to allocate HBM with numactl or memkind library.

3. Quicksort

To parallelize Quicksort with task concept we first defined a limit and number of threads as following:

```
int limit = 300;
int num_threads = 60;
```

Then we set number of threads and parallelized quicksort in the main. We put single to make it only one thread will call first quicksort.

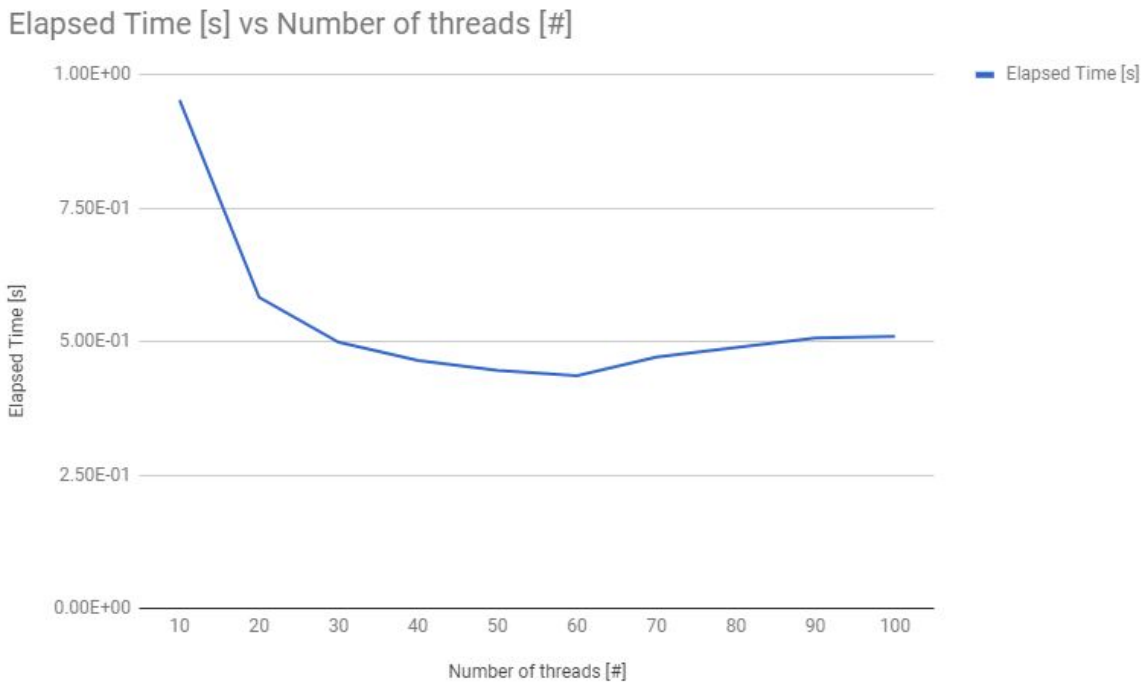
```
omp_set_num_threads(num_threads);
#pragma omp parallel
{
      #pragma omp single
      quicksort(data, length);
}
```

In the recursion, we used task concept and put final clause in order to limit the recursion level otherwise too much granularity causes decreases execution time.

```
#pragma omp task final (right<limit)
quicksort(data, right);
#pragma omp task final (length-left<limit)
quicksort(&(data[left]), length - left);
```

Finally, the strong scalability report is as following (Size of dataset: 10000000):

Number of threads [#]	Elapsed Time [s]
1	7.546245e+00
10	9.523574e-01
20	5.828795e-01
30	4.987238e-01
40	4.645855e-01
50	4.457861e-01
60	4.363345e-01
70	4.712760e-01
80	4.893235e-01
90	5.068484e-01
100	5.100333e-01



4. Matrix-Matrix-Multiplication

On the last assignment our micro-kernel didn't achieve very good performance, so we invested some more time and optimized it. Also regarding our current task in assignment 2. In the following studies, the updated kernel showed a peak performance of 26.16 GFLOP/s.

1.)

For this task, we implemented an algorithm described in [1]. We chose the top branch of Figure 4, which promises good performance and is also favored by the authors if the matrices are stored in column-major order (Section 5.6 of the paper). We developed 4 steps, to divide the big Matrices into smaller problems, which can be parallelized.

GEMM
GEMM divides Matrix A into columns of width K and Matrix B into rows of height K. It then assigns columns i of A and rows i of B to GEPP. (i within 0 and S/K). Rows i of B will already get packed, since the following steps work on this packed version of B.

GEPP
GEPP divides the given columns of A into smaller blocks of the dimension MC x K. It also divides the Matrix C into rows of the dimension MC x S. The given rows of B stay the same for all operations within GEPP. GEPP then assigns block j of A (packed), rows j of C and also the given packed B to GEBP (j within 0 and S/MC).

GEBP
GEBP divides the given block of A into smaller blocks of the dimension M x K, the packed rows of B into smaller blocks of the dimension K x N and the given rows of C into blocks of the dimension M x N and uses the microkernel to calculate these smaller Matrix multiplications.

Micro-Kernel
The Micro-Kernel is the step where the actual computation of the Matrix Multiplication occurs.The Micro-Kernel is used many times, with determined subparts of the big Matrices, to optimize performance and make parallelization possible.

2.)

In the first version of our implementation we just tested the single core performance with only one thread and no parallelization. The task was to assign one pack of A to one team of threads. We solved this, by arranging the algorithm, so that nTeams had to be the same

[1] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw., 34(3):12:1–12:25, May 2008.

number as S/MC. We then said, that each thread in a team should work on one block of height M of MC. These gave us the restrictions:

a.) `threadsPerTeam == MC / M`
b.) `nTeams == S / MC`
c.) `M % 8 == 0` (Micro-Kernel implementation)

Since we only had one team with one thread, we worked with `S = MC = M`, which resulted in bad performance (see Table 4.1).

Single Core Performance (S=MC=M, N=8)

S	K				Microkernel (GFLOP/s)	GEBP (GFLOP/s)	GEMM (GFLOP/s)
32	32				9,52	5,66	5,83
64	64				9,43	9,54	9,30
128	64				9,27	7,49	7,48
256	64				6,79	6,22	5,90
512	64				5,19	4,84	4,80
1024	64				4,09	3,86	3,83

Table 4.1

We then changed the algorithm, to make M independent of threadsPerTeam and tested the new version with different numbers for M, N and K. We found out, that `M = 16, N = 8` and `K = 64` gave us the best performance. That's why we then performed all following tasks with this configuration. MC is always dependent on S and the number of Thread-Teams, since every team should work on one pack of A and the height of one pack of A is MC. If `nTeams == S / MC` wouldn't be true, then we would miss some parts of calculation or calculate to much, which results in a memory violation. That's why `MC = S / nTeams`.

The resulting algorithm has the restrictions:

a.) `(MC / M) % threadsPerTeam == 0`
b.) `nTeams == S / MC`
c.) `M % 8 == 0` (Micro-Kernel implementation)

The resulting algorithm gave us a better single core performance, see Table 4.2.

Single Core Performance (S=MC, N=8)

S	K	M			Microkernel (GFLOP/s)	GEBP (GFLOP/s)	GEMM (GFLOP/s)
32	32	16			26,16	19,99	7,12
64	64	16			26,10	23,02	16,59
128	64	16			25,21	20,19	18,81
256	64	16			25,07	16,57	14,89
512	64	16			23,93	8,93	8,59
1024	64	16			24,14	5,55	5,30

Table 4.2

3.)

We parallelized our implementation, by assigning one pack of A to one team of threads in GEPP. In GEBP we assigned `(MC/M)/threadsPerTeam` parts of MC to one thread in the team. See the GIT Repository for the code.
We tested different combinations of S, nTeams and threadsPerTeam. Regarding the performance of GEMM it showed, that the best performance could be achieved, if there is only 1 Thread per Team. See Table 4.3.

Multi-Thread Performance (K=64, N=8, M=16)							
S	MC	Threads	Teams	Threads per Team	Microkernel (GFLOP/s)	GEBP (GFLOP/s)	GEMM (GFLOP/s)
512	512	16	1	16	24,05	92,91	59,41
512	256	16	2	8	25,11	52,54	77,27
512	128	16	4	4	24,72	36,32	78,48
512	64	16	8	2	25,09	18,93	79,39
512	32	16	16	1	25,28	9,41	96,98
1024	1024	32	1	32	23,79	268,79	105,23
1024	512	32	2	16	24,41	176,79	105,75
1024	256	32	4	8	25,02	92,85	144,32
1024	128	32	8	4	24,70	48,37	155,81
1024	64	32	16	2	25,05	26,95	171,27
1024	32	32	32	1	25,28	12,49	199,54
4096	4096	64	1	64	4,15	218,99	129,29
4096	2048	64	2	32	5,26	185,18	137,48
4096	1024	64	4	16	9,43	117,77	152,47
4096	512	64	8	8	22,66	66,95	153,07
4096	256	64	16	4	22,89	34,35	164,53
4096	128	64	32	2	23,21	19,82	204,42
4096	64	64	64	1	22,96	9,46	232,74

Table 4.3

4.)

In the two strong scaling studies, we tested first, how the performance evolves if we use more threads on a fixed problem size of `S = 4096`. In the second study, we tested, how the performance evolves on a fixed thread number of 64, but with increasing S.
In the previous tasks, we found out, that M=16, N=8, K=64 and one Thread per Team works best. That's why we chose this configuration for the scaling studies.

In the first study (see Table 4.4), we figured out, that with a growing thread number, the performance gets better, but usually not with the same rate how the thread number is growing. From 1 to 2 Threads, we have a 3 times better performance (see last column of Table 4.4). From 2 to 4 Threads, we only have a 2 times better performance. From 8 to 16 Threads, we have a 2,5 times better performance. All other factors are around 1,5 - 1,8. In conclusion, we can say, that more threads increase performance, but if the number of threads double, the performance usually doesn't double.

Strong Scaling (K=64, N=8, M=16) S=4096, increasing Threads

i	S	MC	Threads	Teams	Threads per Team	Microkernel (GFLOP/s)	GEBP (GFLOP/s)	GEMM (GFLOP/s)	GEMM i / GEMM i-1
1	4096	4096	1	1	1	5,30	3,90	3,89	
2	4096	2048	2	2	1	5,42	9,76	11,70	3,0
3	4096	1024	4	4	1	5,92	9,55	23,14	2,0
4	4096	512	8	8	1	22,53	9,86	41,01	1,8
5	4096	256	16	16	1	24,76	9,88	103,09	2,5
6	4096	128	32	32	1	24,58	9,16	153,29	1,5
7	4096	64	64	64	1	23,17	9,51	231,37	1,5

Table 4.4

In the second study (see Table 4.5), we figured out, that for small S, the performance stays roughly the same. From S=1024 to S=2048 and from S=4096 and S=8192 we even got a better performance factor of 1,1 respectively 1,5. For all other cases, the performance was worse with a bigger problem size.

Strong Scaling (K=64, N=8, M=16) 64 Threads, increasing S

i	S	MC	Threads	Teams	Threads per Team	Microkernel (GFLOP/s)	GEBP (GFLOP/s)	GEMM (GFLOP/s)	GEMM i / GEMM i-1
1	1024	16	64	64	1	23,69	10,58	229,84	
2	2048	32	64	64	1	24,65	9,99	262,92	1,1
3	4096	64	64	64	1	23,07	9,42	206,79	0,8
4	8192	128	64	64	1	25,54	10,01	300,51	1,5
5	16384	256	64	64	1	24,62	9,63	134,16	0,4
6	32768	512	64	64	1	10,12	9,67	86,90	0,6

Table 4.5

In conclusion, the best performance was achieved by 64 Threads (also 64 Teams) and a Problem size of S=8192. The GEMM performance was 300,51 GFLOP/s.