This assignment covers shared memory parallelization with OpenMP. With shared memory parallelism we are able to use a single compute node up to its full extent. Together with MPI covered in the third assignment we will have all tools at hand to unleash the complete power of state-of-the-art supercomputers.

1 Shared memory π-calculation (2P)

With $\quad \phi(x) = \dfrac{1}{1+x^2} \quad$ we have $\quad \displaystyle\int \phi(x)\,dx = \arctan(x).$

Hence, π can be calculated through integration of $\phi(x)$: `http://mathworld.wolfram.com/InverseTangent.html`.

1. Develop a serial implementation that integrates function $\phi(x)$ over $[0, 1]$.
2. Parallelize your application using OpenMP. Please implement two different versions using both the *critical directive* and the *reduction clause* in order to ensure the correct summation order.
3. Perform a scaling study of your algorithm.
   - Use `OMP_NUM_THREADS` in order to start your application with different thread counts.
   - Calculate the achieved speed-up and provide interpretations. Please perform weak and strong scaling studies.
     Weak scaling: if you double the number of threads you also double the problem size.
     Strong scaling: You keep the problem size constant but increase the number of threads.

Hint Use the mid-point rule for integration: Split the unit-interval into n equal sized sub-intervals with length $h = \frac{1}{n}$. For each mid-point $\check{x}_i$ of each sub-interval, calculate the value of function $\phi(\check{x}_i)$. Afterwards, sum up all function values. In the end multiply the result with $4 \cdot h$.
Explain why this method works.

2 STREAM benchmark (2P)

The STREAM benchmark `http://www.cs.virginia.edu/stream/` is a popular benchmark to measure the sustainable memory bandwidth of high performance computers.

1. Make yourself familiar with the STREAM Benchmark. Explain the different sub-benchmarks shortly: “Copy”, “Scale”, “Add” and “Triad”.

2. Follow the instructions to adjust the compilation settings accordingly. Set `STREAM_ARRAY_SIZE` to a large value but such that less than 16 GiB of memory are used.

3. Inform yourself about KNL processor modes [4, 2]. Explain `flat` and `cache` mode. Inform yourself about *non-uniform memory access (NUMA)* and explain the difference between `quadrant` and `snc4` mode.

4. Write job scripts for the stream benchmark with processor modes $\{\texttt{cache}, \texttt{flat}\} \times \{\texttt{quad}, \texttt{snc4}\}$, run them on the cluster [1], and report achieved bandwidths.

   Hint: See the SLURM documentation [6].

   Switching cluster modes is currently broken. I created a ticket and I am going to inform you during the first week about the status of this and the next question. In any case you should be able to test flat vs. cache mode.

5. Why is the bandwidth in flat mode worse than in cache mode? Use `numactl` or the library `memkind` to mend this for both `quad` and `snc4` modes.

3 Quicksort (2P)

1. Parallelize the quicksort implementation given in `quicksort.c`. Please employ the task-concept of OpenMP 3.1. Use the `final` clause for stopping the parallelization of the recursion at a sufficient level of the recursion.

2. Examine the scalability (strong scaling) for different problem sizes and plot your results.

4 Matrix-Matrix-Multiplication (4P)

On the last assignment we focused on writing a small micro-kernel for small matrix-matrix-multiplications. Here, we want to embed this kernel into a large matrix-matrix-multiplication of the form

$$C_{mn} = \sum_{k=0}^{S-1} A_{mk} \cdot B_{kn} \,, \qquad m = 0, \ldots, S-1, \quad j = 0, \ldots, S-1.$$

for the matrices $A \in \mathbb{R}^{S \times S}, B \in \mathbb{R}^{S \times S}$, and $C \in \mathbb{R}^{S \times S}$, where S is large. A reference implementation is given in `dgemm.c`.

1. Implement `GEBP`, `GEPP`, and `GEMM` according to [8]. Use the algorithm the authors argue to be best, i.e. by always taking the top branch in Figure 4. See also Figure 8.

   Use your micro-kernel from the last assignment sheet for the function `microkernel`. (You might need to change the stride for the C matrix in your micro-kernel.) If your micro-kernel did not achieve good performance, you may also generate a micro-kernel using LIBXSMM [3].

Verify that your implementation is correct.

Hint: For LIBXSMM compile with `make generator` *and use the gemm generator in the binaries folder.*

2. Determine suitable values for $\mathtt{m_c}, \mathtt{k_c}, \mathtt{m_r}, \mathtt{n_r}$ and give arguments for your choice in your report. Test the single-core performance of your code.

   Hint: Have a look at Section 6 in [8]. Note that $\mathtt{M} = \mathtt{m_r}, \mathtt{N} = \mathtt{n_r}, \mathtt{K} = \mathtt{k_c}$.

3. Parallelise your implementation using OpenMP.

   The first level of parallelism should be introduced in `GEPP`, where a team of threads is assigned to work on one `GEBP`. (They all work on the same pack of `B`.)

   Use the threads in the thread team to parallelise `GEBP`. Each team has one pack of `A`.

   Ensure that teams are spawned correctly and that in each team `threadsPerTeam` threads are working on a `GEBP`. Furthermore, verify that your implementation is still correct.

   Which number of threads per team gives you the best performance?

   Hint: Your implementation could employ OpenMP nested parallelism.

4. Perform two strong scaling analyses of your final implementation. In one analysis you should increase the number of threads on a fixed problem size. In the other, fix the number of threads to 64 and increase the problem size.

   Which combination of problem size and number of threads gives you the highest performance?

   General hints:

   - You do need not to handle boundary cases. Instead, choose $\mathtt{S} = \mathtt{n} \cdot \mathtt{S}'$, such that $\mathtt{S}'$ is divisible by $\mathtt{m_c}, \mathtt{k_c}, \mathtt{m_r}$, and $\mathtt{n_r}$.
   - Leave the kindergarten: Try problem sizes $\mathtt{S} > 4000$.

Deliverables

The following deliverables have to be handed in no later than 08:00 AM, Monday, 20 November, 2017. If there is no submission until this deadline, the exercise sheet is graded with 0 points. Small files (<1 MB in total) can be send as an attachment directly to *uphoff AT in.tum.de* and *chaulio.ferreira AT tum.de*. Larger files have to be uploaded at a place of your choice, e.g. `https://gitlab.lrz.de/`, `http://home.in.tum.de/`, `https://syncandshare.lrz.de`. In either case inform us about the final state of your solution via e-mail.

- A short report which describes your work and answers all questions in this assignment.
- All of your code.

- Slides for the presentation during the next meeting.
- Output of all runs. Figures (e.g. scaling graphs) if applicable.
- Documentation how to build and use your code.

Literature

[1] CoolMUC3 features. `https://www.lrz.de/services/compute/linux-cluster/batch_parallel/specifications/index.html#features`. Accessed: 2017-11-05.

[2] KNL processor modes. `http://www.nersc.gov/users/computational-systems/cori/configuration/knl-processor-modes/`. Accessed: 2017-11-05.

[3] LIBXSMM. `https://github.com/hfp/libxsmm`. Accessed: 2017-11-05.

[4] Memory modes and cluster modes. `https://software.intel.com/en-us/articles/intel-xeon-phi-x200-processor-memory-modes-and-cluster-modes-configuration-and-use-c` Accessed: 2017-11-05.

[5] OMP tutorial. `https://computing.llnl.gov/tutorials/openMP/`. Accessed: 2017-11-05.

[6] SLURM KNL guide. `https://slurm.schedmd.com/intel_knl.html`. Accessed: 2017-11-05.

[7] Thread affinity interface. `https://software.intel.com/en-us/node/522691`. Accessed: 2017-11-05.

[8] 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.