Session 3: NUMA and SIMD

Christian Terboven, Paul Kapinos
IT Center, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, kapinos}@itc.rwth-aachen.de

Yun (Helen) He
NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
yhe@lbl.gov

Abstract

This document guides you through the exercises. Please follow the instructions given during the training session.

The prepared makefiles provide several targets to compile and execute the code:

- debug: The code is compiled with OpenMP enabled, still with full debug support.
- release: The code is compiled with OpenMP and several compiler optimizations enabled, should not be used for debugging.
- run: Execute the compiled code. The `OMP_NUM_THREADS` environment variable should be set in the calling shell.
- clean: Clean any existing build files.

The provided *.slurm is a batch script that you can use to submit a batch job to run on a Perlmutter compute node. Submit the job via command "sbatch *.slurm", and check the output file after it is run, pay attention to run time using different number of threads too.

You can also run an interactive batch job via "salloc" to get on a compute node. For example:

```
% salloc -N 1 -q interactive -C cpu -t 30:00
<will land on a compute node>
%  export OMP_NUM_THREADS=8     (please try a few different values)
% ./mycode.exe
```

Please refer to https://github.com/NERSC/openmp-series-2024/blob/main/Session-1-Introduction/Using-OpenMP-Compilers-on-Perlmutter-CPUs-May2024.pdf for more details on Using various OpenMP compilers on Perlmutter and running jobs.

1 Parallelization of Pi (numerical integration)

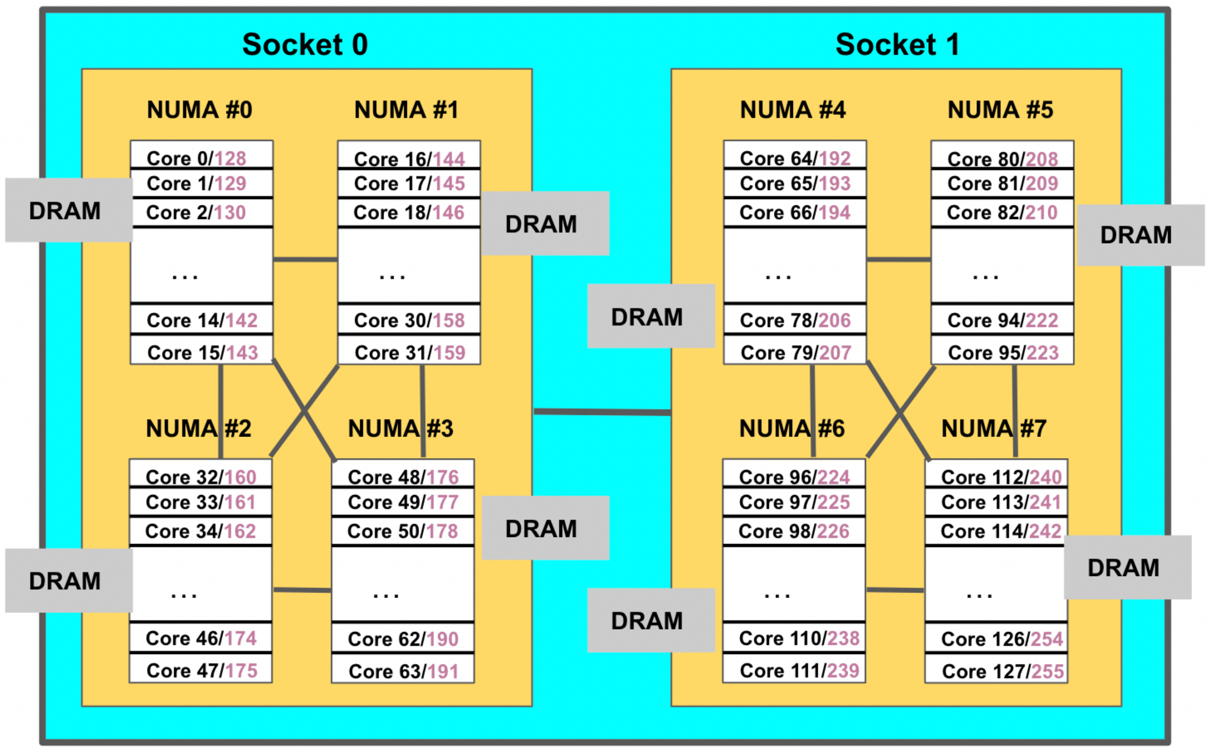
Go to the `pi` directory. This code computes Pi via numerical integration. Compile the `pi` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where *procs* denotes the number of threads to be used.

Exercise 1: Vectorize the Pi code with OpenMP SIMD. The compute intensive part resides in one single loop in the `CalcPi()` function. Re-compile and execute the code in order to verify your changes. Try to use different compilers and different optimization levels to compare performance results.

2 Understanding Thread Affinity

Go to the `xthi` directory.

Exercise 1: Follow the instructions and steps in README.xthi to get the hardware information of the Perlmutter CPU compute node, then compile both `xthi.c` and `xthi_nested_omp.c` codes. Understand the `numactl -H` output by referring to the node diagram below:



Exercise 2: Compile and run both `xthi_.c` and `xthi_nested_omp.c` using different `OMP_NUM_THREADS`, `OMP_PROC_BIND` and `OMP_PLACES` and try to understand the results of thread affinity, i.e., binding of OpenMP threads to the logical CPUs on the compute node.

3 The Importance of First Touch

Go to the `stream` directory. Follow the instructions and steps in `README.stream`.

Exercise 1: Compile and run `stream_nft.c` code with the provided `run_stream_nft.sh` with different `OMP_NUM_THREADS`, `OMP_PROC_BIND`, and `OMP_PLACES` settings, and check the STREAM Triad bandwidth results.

Exercise 2: Modify `stream_nft.c` code so that it does first touch, and run the same experiements as above. Check the the TRIAD memory bandwidth results, and compare them with the no first touch results. Explain why doing first touch helps with STREAM memory bandwidth results.

4 Parallelization of an iterative Jacobi Solver

Go to the `jacobi` directory. Compile the `jacobi.c` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

Exercise 1: Parallelize at least the most compute-intensive program part with OpenMP, or continue from where you left off in Session 1.

Exercise 2: Experiment with different thread affinity binding policies (`OMP_PROC_BIND` environment variable).

# Threads	Binding policy	Runtime [sec]	Speedup
1			
2			
4			
8			
16			
24			

Exercise 3: Review your parallelization and ensure that data is correctly laid out on NUMA. Execute your experiments again. Why is there a difference, or maybe where is there no difference?

# Threads	Binding policy	Runtime [sec]	Speedup
1			
2			
4			
8			
16			
24			