M. Troyer, P. Hadjidoukas
ETH Zentrum, ML H 44
CH-8092 Zürich

Fall semester 2015

Set 9 - Diffusion and MPI

Issued: November 25, 2015

Question 1: Diffusion in 2D with MPI

If we want to spread the heat beyond a single node we need message passing. In this exercise we will parallelize the 2D diffusion equation with MPI.

a) Parallelize your serial 2D diffusion code from Ex 2 with MPI. Use the simple domain decomposition scheme that is described in the lecture notes (i.e. distribute the rows evenly to the MPI processes).

The parallel code can be found in `diffusion2d_mpi.cpp`. To minimize communication overhead we compute the interior of the local domain while waiting for the boundaries by using non-blocking asynchronous communication.

b) Suggest other ways to divide the real-space domain between processes with the aim of minimizing communication overhead.

Another approach is to split the grid into square tiles with one tile per process. The total amount of communication per process if the domain is divided into tiles is $2[N_x/P_x+N_y/P_y]$ and for strips along the y-axis $2N_x$, where $N_{x,y}$ and $P_{x,y}$ is the number of grid points and processes in each direction. As we are doing sparse matrix-vector multiplication the computation scales as N_xN_y/P_xP_y. Hence finite difference discretization should scale much better if we use tiles rather than strips.

For the boundaries, instead of sending and receiving on each edge of the sub-domain we can send twice the data on one side per dimension and shift the local grid. The next time-step we send from the opposite side and shift the grid back. This will reduce communication overhead and can have a noticeable effect for a small local domain.

c) Compare the performance to your previous implementation using shared memory within a single node (up to 24 cores). Use the wall-time needed to propagate the system a fixed number of time-steps as a measure. *hint: remember to synchronize the clock*

In Figure 1, we observe that both implementations exhibit comparable performance on 12 cores for 4 different problem sizes. MPI is slightly slower than OpenMP on the 3 larger cases, as the communication cost is higher than the runtime overheads of the OpenMP parallelization. As the comparison is performed on a single NUMA node, the effect of memory bandwidth is similar for both cases. On 24 cores, however, the MPI implementation is significantly faster. This is attributed to the better memory locality of the MPI code, as the MPI processes automatically allocate memory for their arrays on the local NUMA node. In contrast, the allocation of the vector in the OpenMP implementation is not NUMA-aware.

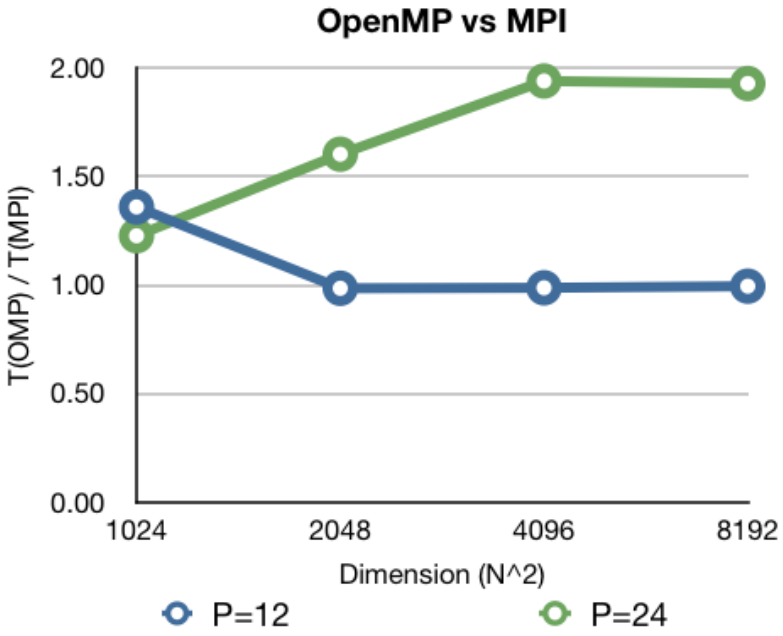

Figure 1: Ratio between execution times of OpenMP and MPI for 100 time-steps.

d) Make a strong and weak scaling plot up to 48 cores.

Looking at the strong scaling (Fig. 2), for a total grid size $N = 2^{10}$ the sub-matrix fits entirely into the cache minimizing memory access and resulting in almost perfect scaling. We observe similar behavior for $N = 2^{11}$, as soon as the application utilizes the second NUMA node. As we increase the system size we exceed the cache size and additional memory accesses reduce performance. Despite the inter-node communication between the two nodes, the code continues to scale on up to 48 processes (cores).

In the weak scaling (Fig. 3) the total computational domain per process is kept constant. More specifically, the global grid dimension N_g is equal to $N\sqrt{(P)}$, where P is the number of processes and N is the grid dimension when running on a single core ($P = 1$). The drop in the scaling for 12 cores is attributed to the fact that the problem size fits entirely into the cache for $P = 1$. We also observe that the efficiency remains constant as the number of NUMA nodes increases.

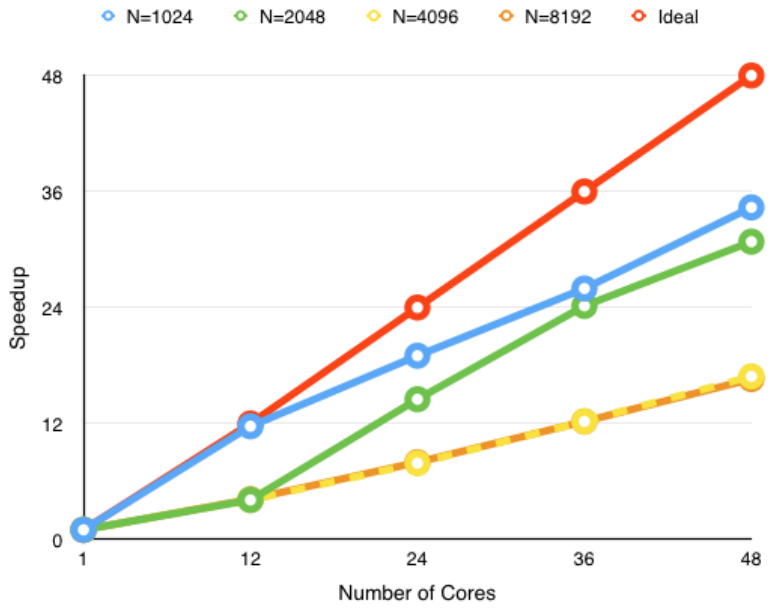

Figure 2: Strong scaling for 100 time-steps. N is the global grid dimension

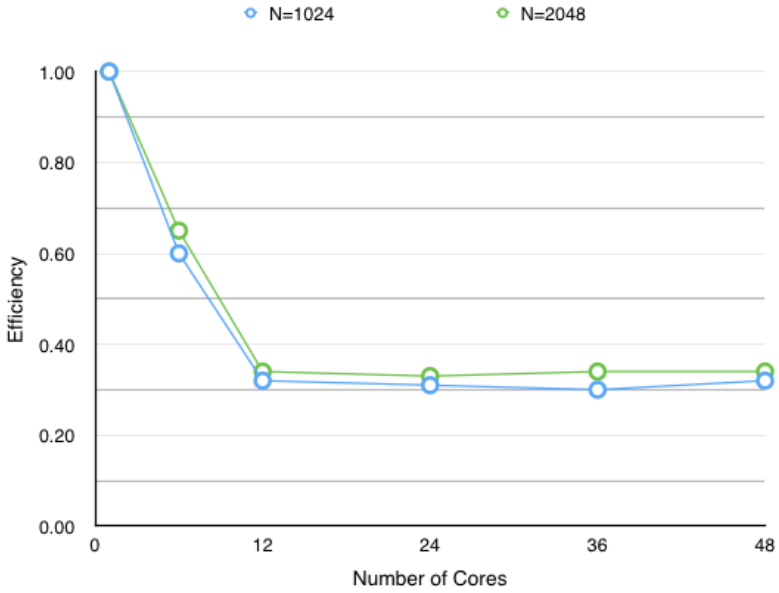

Figure 3: Weak scaling for 100 time-steps. N^2 is the number of local grid points