Sheet 5

Your tasks: Add loop-level parallelism using openMP to this code, and assess whether any of the expected benefits, listed below, can be reached with the methods that you are knowledgeable with? The methods you learnt during Programming Parallel Computers course are sufficient. Provide evidence in the form of tables, plots, … , and present a short analysis on each point.

1. reduction in memory usage? **(1/3 of 150 points)**
2. performance increase? **(1/3 of 150 points)**
3. extended scale up? **(1/3 of 150 points)**

Objectives: add thread parallelism using openMP most importantly to the interior (green points) and possibly to the edge (red point) computations.

How? We modify the pure MPI code to:

- **Allow threading using the FUNNELED option for MPI; as we will have only one MPI process making the MPI calls, this support level is adequate.**
- **All the variables in the stencil update loops are independent, hence it is safe to simply add "#pragma omp parallel for" sections around the outer loops.**
  - Somewhat more sophisticated variants were also investigated
    - (1) #pragma omp parallel for … #pragma omp simd for inner loop for interior updates
    - (2) #pragma omp parallel for collapse2()
  - The best performance was found with option (1)
  - Some student solutions proposed using #pragma omp parallel for both in the outer and inner interior loop, *but this approach is fundamentally flawed*. In case you proposed such a solution, please repeat PPC Chapter 3 materials.

What are our expectations? - a theoretical approach

During the course we have gained enough theoretical background to form an assessment of the situation without performing any numerical experiments/do any coding yet. The MPI parallelization in stencil codes bases on dividing the work between different processes so that each processing element (hereafter PE) executes the same code in SPDM manner. In the case of the PEs residing on different nodes they need to additionally communicate to be able to perform the computations, as only the green cells can be updated without sharing data with neighboring PEs. The extra time consumed by these communications has the potential to slow down the performance. Hence, compute and communication should be overlapped, so that these would occur concurrently. MPI libraries allow for concurrency through the non-blocking communication functions. Optimally, the communications would take a shorter time to complete than the computations, when we could always guarantee parallel computations by the different PEs. This ideal regime we called the compute bound regime in the lecture materials.

Now the plan is to add openMP threads in the hope for more parallelism in performing the double loops for green (and possibly red) cell stencil updates. Ideally, this would result in the computations to take place even faster, while the communications would still be performed in the same way as before. By assigning more threads to the computations and less MPI tasks, one could also try to optimize for the memory consumption and communication speed.

1. The total memory required per computation is set by the total number of grid points (times physical variables, times the states required for time integration), while the size of additional buffers for communication is set by the stencil halos of the current state of physical variables. For 2D 2nd order von Neuman stencil, the memory requirements for both computation and communication are modest, see Table 1. For an already totally oversized computation w.r.t. the smooth initial condition (10,000 cells in one direction) one would need around 1,526GB memory in total (assuming that two different time states of the main array (temperature field) needs to be stored is a double precision in each grid point), while the halo zones would constitute of the order of half to a few MB depending on the number of nodes used; see also the tabulated values in Table 1 for this setup in the strong scaling scenario. For a weak scaling scenario the increase of memory due to the halo buffer size is even smaller fraction of the total memory consumption.

   Comparing actual measurements of the memory consumption of the runs using tools such as seff/sacct/sstat can be made. In the following measurements,

   sstat --format=MaxRSS,AveRSS,AveVMSize,JobID -j <jobid>

   at runtime was used, and the MaxRSS value was used as an estimate of the total memory consumption. This is naturally an upper limit, but gives you the worst scale scenario.
   A similar measurement could be made with

   sacct -j <jobid> --format="MaxRSS,AveRSS,AveVMSize,JobID"

   Note that MaxRSS values are computed per task, hence jobs with multiple tasks should take this into account.

nproc	Yellow	Red	Green	Ratio Y/G	Mem per core [MB]	Tot halo mem [MB]	Tot mem [MB]
1	-	-	100000000	-	1525,88	-	1525,88
2	30000	29996	49985001	0,00060018	762,94	0,46	1526,79
4	20000	19996	24990001	0,00080032	381,47	0,61	1527,10
8	15000	14996	12492501	0,00160128	190,73	0,92	1527,71
16	10000	9996	6245001	0,00320513	95,37	1,22	1528,32
32	7500	7496	3121251	0,00642053	47,68	1,83	1529,54
64	5000	4996	1560001	0,01288231	23,84	2,44	1530,76
128	3750	3746	779376	0,02593085	11,92	3,66	1533,20

Table 1: Theoretical estimation of memory consumption in the case of 10,000 cells in each direction using a strong scaling scenario. Note: periodic case assumed, so not totally accurate numbers, but at least tracing the maximum memory usage. Also, Euler time stepping requires the storage of two states simultaneously in memory, hence the mem in MB estimates include a factor of two w.r.t. green points.

   From Table 2 we can see that MaxRSS values indicate that the memory consumption increases quite strongly when the number of MPI tasks is increased, while remains constant with threading only. **To optimize the memory usage, the number of threads should be maximized versus the number of MPI tasks.**

2. **and 3.** From **Table 1** we can also see that the number of edge (red) cells per process is small in comparison to the interior (green) points. This means that threading the computation of the green cells will give us the most of the parallelization gain. The computation of the red

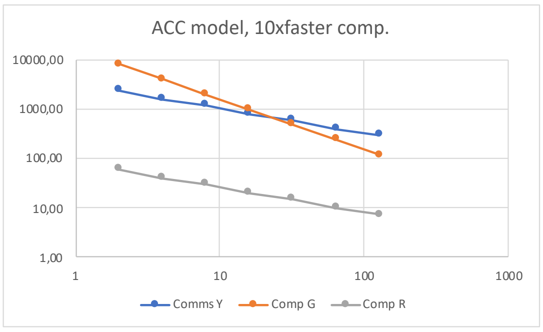
Total number of PEs	nodes	ntasks	cpus	MaxRSS [MB]	Ratio (to serial)
1	1	1	1	2757	1
2	1	1	2	2757	1
2	1	2	1	3952	1,4
2	2	1	1	3952	1,4
4	1	4	1	6344	2,3
4	1	1	4	2757	1
4	2	1	2	3952	1,4
8	2	4	1	11128	4,1
8	1	8	1	11128	4,1
8	1	1	8	2757	1
8	1	2	4	3952	1,4
8	2	1	4	3952	1,4
16	4	4	1	20704	7,5
16	1	1	16	2757	1
16	1	4	4	6344	2,3
32	1	1	32	2757	1
32	1	32	1	39904	14,5
32	4	1	8	6344	2,3
32	4	8	1	39904	14,5

points is done in four different sections with loop structure within each; these computations cannot be done concurrently w.r.t communication. Here, openMP parallelism could also be useful in principle, but no large gains can be expected due to the low order of the stencil.

Table 2: Memory consumption of 10,000 x 10,000 simulation grid with varying number of nodes, tasks and threads. Estimates are based on MaxRSS values, and give an upper limit.

Let us now consider the case when loop parallelization with openMP is added to the computation of the interior (green) and edge (red) cell updates. Using loop level parallelism with openMP alone (**pure openMP case**; specifying only one MPI process and one node) should be roughly equivalent of dividing the work to different MPI processes using no threading, as the end result of both approaches is that different PEs perform the computation of interior points in parallel. This equivalence assumes that the MPI communications can be overlapped with communication, but this we can enforce by using large problem sizes, that will give the processes enough to compute versus communicate. Hence, we are not expecting to see a performance increase from swapping from pure MPI-parallelism to pure loop parallelism with these assumptions.

What about the hybrid implementation? Adding threads to compute loops in parallel means that this part of the code will be accelerated between the different MPI processes. The MPI communication will, however, take equal amount of time as with the same amount of MPI processes without thread parallelism. We make computations faster, while the communication still happens with the same speed. This means that keeping on adding threads will only help until the communication bound limit is reached. Such scenarios are illustrated in **Figures 1,2, 3,** and **4**. In these figures we plot the estimated time spent in communicating the halo zones (Y for yellow, blue symbols and lines), and in computations of the interior (G for green, orange symbols and lines) and edge (R for red, gray symbols and lines) cells. In **Figures 1** and **2** we have compared a small problem size (400x400 grid points) in strong scaling scenario using the ACC model with different efficiencies of computation and communication. On the left hand side plot, the communications are 10 times slower and on the right hand side 20 times slower; the left side could resemble a situation not accelerated with added threading withing the interior loop, and the right side with threads added. Due to the small problem size, the problem becomes communication bound at 32 MPI processes, and after that point, the scaling is dominated by the communication, and will cause a degradation of the strong scaling performance. In this situation, adding threads will only make the situation worse. The computations will be done even faster, hence the communication bound limit is reached even earlier. In other words, a promising performance increase in the beginning will degrade sooner due to the latency related to communications that become dominant earlier.



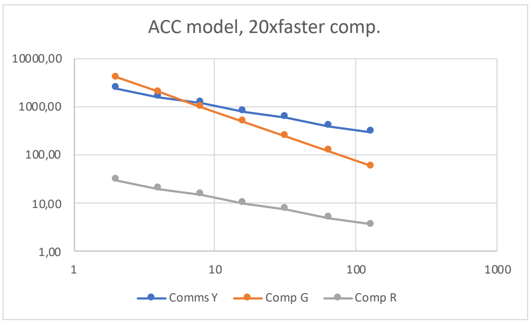
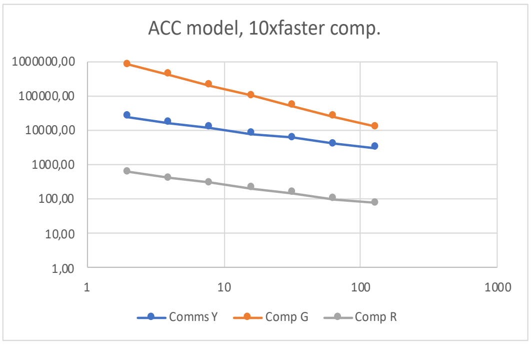


Figure 1: Small problem size

Figure 2: 5 with comp. accelerated

If we increase the problem size to 4,000 x 4,000 cells, then this problem is somewhat alleviated, or pushed towards a larger process count, as is illustrated in **Figures 3** and **4**. As there are more computations to perform overall w.r.t. communication, then the communication bound limit is reached later in terms of the MPI processes, and acceleration with more threads can lead to performance improvements up to a higher number of MPI processes. Again, the communication bound limit is, however, reached earlier. Hence, for a problem size large enough, we may expect performance increase but not a trivial extension of the scaling up to a higher number of processors. With the limited resources and we had in Triton for this course, it might be difficult to quantify these effects, however.



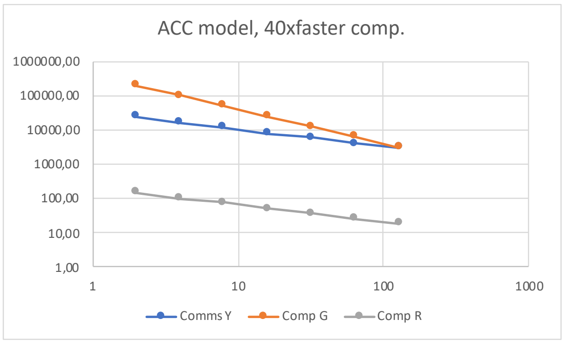


Figure 3: Large problem size

Figure 4: 7 with comp. accelerated

Practical approach: measuring the performance and scaling-up of the different code versions

Strong scaling strategy:

We will keep the problem size and number of integrations fixed, and double the amount of processors. Chosen problem size was 4,000x4,000 grid points, and 1,000 integration steps. Time was measured using the already implemented timing in the original code, using the **MPI_Wtime** function. Measurements are done around the integration loop, including the image and checkpoint writing, hard-coded to take place each 500 and 200 time units. To get results that are totally independent of I/O performance, one should increase these numbers above the integration time length, as was done in the above tests.

2.-3.1. Pure MPI case

Two different types of experiments were made:

1) Always running as many jobs in the same node as possible (Increasing --ntasks-per-node, keeping --nodes=1 as long as possible, and then starting to increase it when 24 core limit was reached)
2) Forcing them to run in different nodes (possible up to --nodes=16).

Data from these experiments are presented in **Table 3** and **Figure 5**, where we list and plot the data (blue) in the same format as in the lecture material, where speedup according to Amdahl's law was expressed as sequential time over time with *P* processes. The first observation is that we see different kind of behavior in 1) and 2)-type experiments: when the processes are forced to run on different nodes (2), the performance of the code shows somewhat but consistently better speed-up. When the MPI processes are distributed over as many cores within a node as possible (1), the performance is worse, and scaling is worse when the processor code is low-intermediate. When the processor count is increased even more, the experiments seem to converge in speed-up and performance. Overall, the speedups observed with either method are excellent up to 16 processors, and then deviations from the ideal scaling start to become visible. The scale-up is still decent up to 128 processors, and then a sharper degradation is seen for 256 processors. One could attempt a fit to Amdahl's theoretical law to obtain the serial fraction, but due to the fluctuations seen in the numerical tests, no absolutely certain number can be achieved. Both types of experiments, nevertheless, fit Amdahl's law with a serial fraction of only *f*=0.4%, shown in **Figure 5** with an orange plot. Hence, this code has a tiny serial fraction, which we could have already anticipated from the theoretical considerations.

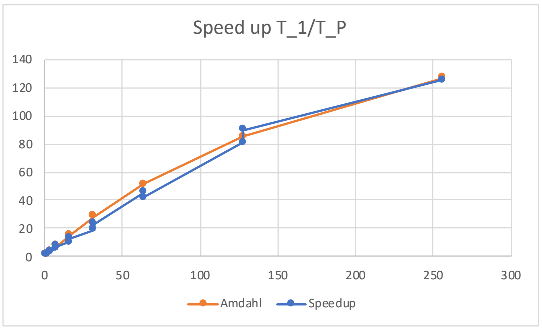


Figure 5: Speed-up according to Amdahl's law

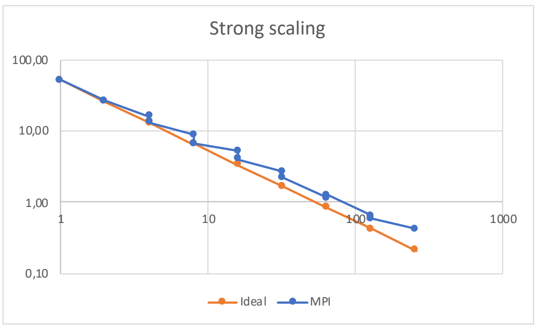


Figure 6: Strong scaling as halving of time when doubling processor count

Total number of tasks	Nodes	CPUs per node	MPI time	Ideal	Amdahl T_1/T_P
1	1	1	51,49	51,49	1
2	1	2	26,20	25,75	1,9654817
2	2	1	25,97	25,75	1,98304236
4	1	4	15,93	12,87	3,23292803
4	4	1	13,18	12,98	3,90680559
8	1	8	8,77	6,49	5,86948592
8	8	1	6,63	6,59	7,76471475
16	1	16	5,16	3,30	9,97346504
16	16	1	4,01	3,32	12,8283508
32	2	16	2,66	1,65	19,3631236
32	16	2	2,18	1,66	23,577381
64	16	4	1,13	0,83	45,6498227
64	4	16	1,24	0,83	41,6947368
128	16	8	0,64	0,41	81,0062926
128	8	16	0,57	0,41	89,8656195
256	16	16	0,41	0,21	124,881973

Table 3: Strong scaling results of pure MPI case. Total number of processes, nodes, and CPUs per node, measured time is seconds, compared with ideal scaling and Amdahl's law with *f*=0.4%

There is also another, more common (read: if you are asked to present the scale-up of your code when applying for time in supercomputers), way of presenting the scaling results, namely computing the speedup as the ratio of times between doubling the processors (**Table 3** columns MPI time, Ideal and **Figure 6**). In this figure, the ideal scaling is presented with an orange curve, and different measurements with blue circles and lines. Type 2) experiments form a consistent, linear trend that deviates slightly from the ideal curve, and only for the highest processor count of 256 starts seriously deviating from the trend. Type (1) experiments deviate from this linear trend for intermediate processor counts, forming a "hump" towards degraded performance. There are some candidates for causing such an intranode performance degradation:

- MPI functions are not optimized well for shared memory
- There is a bandwidth limitation when too many processes are mapped on the same socket/node.
- Some other sub-optimal mapping issue with the processes onto the sockets/nodes.

We can examine the first option using pure openMP parallelization.

2.-3.2. Pure openMP case

Next we disable MPI and investigated the purely threaded version. We use a problem size of 4,000x4,000, and run and time "pure" openMP (one MPI process and several threads) and compare those with the results presented in 2.1. with the same setup with pure MPI. We again enforce Haswell nodes through SLURM (#SBATCH --constraint=hsw), and use "export OMP_PROC_BIND=TRUE" for enabling thread affinity as is recommended in the Triton manual.

nodes	threads	time
1	1	51,53
1	2	26,28
1	4	14,51
1	8	9,08
1	16	5,46

Table 4: Pure openMP timings

The timings are presented if **Table 4**, and the first observation is that the openMP version runs nearly equally fast as the intranode pure MPI. Similar degradation of performance is seen when large amount of threads within a node is used, which rules out this being due to the non-optimal MPI functions for shared memory. This also confirms our other hypothesis of the edge cell update parallelization being of minor importance.

2.-3.3. Hybrid case

The hybrid version also shows large fluctuations in the performance (see **Table 4** and **Figure 9**), depending on how the MPI processes and threads are distributed. The optimum seems to be to distribute the processes as “widely” as possible, using maximally 4-8 threads within a node. Usage of more threads seems to often degrade the performance. Following these constraints, the hybrid version is somewhat more performant than the MPI version especially at the low and high process count range. Even at the largest process count, the hybrid version is still performing better, but shows similar sharper degradation as the pure MPI version already. Even higher number of processes should be benchmarked to confirm this. **Hence, the answers to questions are**

- **Task 2: yes, performance gains can be achieved, but a lot of work to optimize the node/thread distribution is required, possibly even harder than here. If you distribute badly, the performance can be degraded.**
- **Task 3: plausible, but not confirmed with the resources and time available.**

Total number of PEs	MPI	Thread	Time	Configuration
1	1	1	51,68	
2	1	2	25,68	
4	2	2	13,12	sep nodes
4	2	2	14,42	
8	2	4	7,28	
8	4	2	7,62	
8	4	2	6,88	sep nodes
16	2	8	4,26	sep nodes
16	4	4	4,52	sep nodes
16	8	2	4,66	sep nodes
16	2	8	5,07	
16	4	4	5,48	

16	8	2	5,15	
32	8	4	2,00	sep nodes
32	4	8	3,51	sep nodes
32	2	16	2,77	sep nodes
32	16	2	2,65	sep nodes
32	2	16	2,86	
32	16	2	2,65	
64	16	4	1,40	8,2,4
64	8	8	1,51	4,2,8
64	16	4	1,36	4,4,4
64	16	4	1,57	16,1,4
64	16	4	1,44	8,4,2
128	8	16	0,57	8,4,4
256	16	16	0,41	16,4,4

Table 5: Hybrid code timings. In the last column we indicate the node/MPI task/thread configuration used.

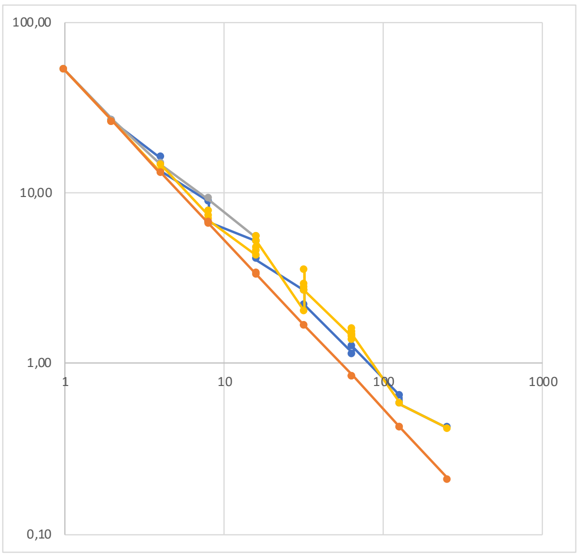


Figure 9: Results from all strong scaling tests collapsed together: orange symbols and line: ideal strong scaling; yellow: hybrid experiments; blue: pure MPI from Table 3.

Procs	Nodes	Cores per nodes	Size	MPI time
1	1	1	1000x1000	8,52
2	1	2	2000x1000	9,36
2	2	1	2000x1000	9,24
4	1	4	4000x1000	9,18
4	4	1	4000x1000	9,73
4	2	2	4000x1000	9,02
8	2	4	8000x1000	7,82
8	4	2	8000x1000	6,74
16	4	4	16000x1000	8,33
16	8	2	16000x1000	10,92
16	2	8	16000x1000	10,00
16	16	1	16000x1000	10,86
16	1	16	16000x1000	10,88
32	2	16	32000x1000	11,33
32	16	2	32000x1000	11,00
32	8	4	32000x1000	8,29
32	4	8	32000x1000	10,55
64	4	16	64000x1000	12,92
64	16	4	64000x1000	13,39
64	8	8	64000x1000	12,76
128	16	8	128000x1000	18,46
128	8	16	128000x1000	12,96

Table 6: Weak scaling results: Total number of processes, nodes, and MPI tasks per node; measured time in seconds.

Example of performing a weak scaling test:

We based our performance and scale-up tests on strong scaling measurements. For completeness, we also discuss how weak scaling performance is measured using the pure MPI version. Here we wish to fix the problem size/process. We start with 1,000x1,000 points, and keep on doubling the number of processors and the grid size in same proportion in one direction, see **Table 6**. In **Figure 10**, we plot these results. We notice that weak scaling can be maintained longer than strong scaling - only points with the highest number of processors (64 and 128) show deviations from the ideal law. The intranode degradation seen in the strong scaling experiments is not evident in this set of runs.

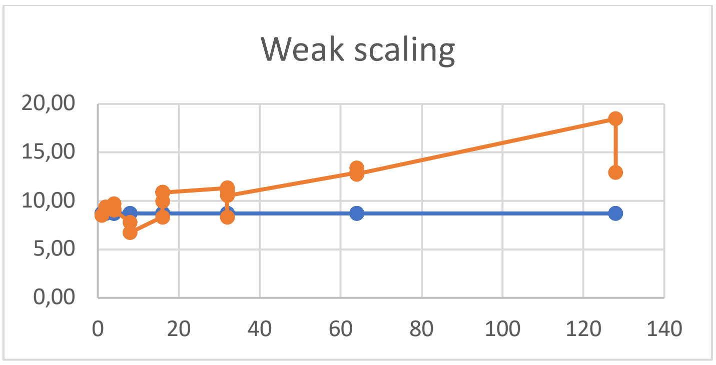


Figure 10: weak scaling of the pure MPI code. Blue ideal, orange measured scaling with the pure MPI code.