CS-E4690 Programming Parallel Supercomputers
Sheet 5 Report
Nguyen Xuan Binh 887799

Hybrid CPU computing

1. Introduction to the physical case

Now you know enough to run, autopsy, and evaluate a real application, which also does meaningful computations, not only communications. Let us investigate one of the most simple, but very important types of partial differential equations in physics and nature – the diffusion equation, in our case applied to describe diffusion of heat. For overall details, you can even start from https://en.wikipedia.org/wiki/Diffusion_equation, and virtually every textbook in physics covers this type of equation to varying detail.

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u,$$

where **u(x, y, t)** is the temperature field that varies in space and time, and α is thermal diffusivity constant. The two dimensional Laplacian can be discretized using finite differences that form a second order von Neuman stencil:

$$(\nabla^2 u)(i,j) = \frac{u(i-1,j)-2u(i,j)+u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1)-2u(i,j)+u(i,j+1)}{(\Delta y)^2}.$$

Given an initial condition (u(t=0) = u0) one can follow the time dependence of the temperature field with explicit time evolution method:

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t \alpha (\nabla^2 u)^m(i,j),$$

where Δt is the length of the time integration step. For a unique solution, boundary conditions are needed, which can be specified as conditions for temperature, its normal derivative or a combination of both.
We will start with an MPI implementation of a two-dimensional (2D) version of the heat equation. The two dimensional grid is decomposed along both dimensions, and the communication of boundary data is overlapped with computation. Restart files are written and read with MPI I/O. See code usage markdown instructions (CODE_USAGE.md) for more details on how to run the code.

Your tasks
==========

Add loop-level parallelism using openMP to this code, and assess whether any of the expected benefits, listed below, discussed in the Lecture 5 materials, can be reached with the methods that you are knowledgeable with? The openMP 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?

2. performance increase?

3. extended scale up?

Both your code implementation and the results you collect in a short pdf description of the exercise project will be evaluated (please name it as report.pdf).

The place that I add the loop-level parallelism to corresponds to the green points given in Lecture 5 slides

Typical example: ISLs using MPI Loop-level parallelism to be added

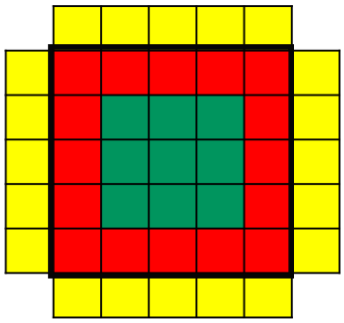
Repeat:

Initiate communication of yellow halos;

Do update of the green zones;

Wait for communications to finalize;

Update the red zones;



Aalto University
School of Science

which is the update of the interior points. The OpenMP directive #pragma omp parallel for is used to parallelize the loop, which distributes the computation of the interior points across multiple threads.

```
/* Update the temperature values using five-point stencil */
void evolve_interior(field *curr, field *prev, double a, double dt)
{
    int i, j;
    int ic, iu, id, il, ir; // indexes for center, up, down, left, right
    int width;
    width = curr->ny + 2;
    double dx2, dy2;

    /* Determine the temperature field at next time step
     * As we have fixed boundary conditions, the outermost gridpoints
     * are not updated. */
    dx2 = prev->dx * prev->dx;
    dy2 = prev->dy * prev->dy;

    // Parallelize the update over the interior points using OpenMP
    #pragma omp parallel for private(i, j, ic, iu, id, il, ir)
shared(curr, prev, width, dx2, dy2)

    for (i = 2; i < curr->nx; i++) {
        for (j = 2; j < curr->ny; j++) {
```

```
            ic = idx(i, j, width);
            iu = idx(i+1, j, width);
            id = idx(i-1, j, width);
            ir = idx(i, j+1, width);
            il = idx(i, j-1, width);
            curr->data[ic] = prev->data[ic] + a * dt *
                                ((prev->data[iu] -
                                  2.0 * prev->data[ic] +
                                  prev->data[id]) / dx2 +
                                 (prev->data[ir] -
                                  2.0 * prev->data[ic] +
                                  prev->data[il]) / dy2);
        }
    }
}
```

The private(i, j, ic, iu, id, il, ir) part ensures that each thread has its own copy of the loop indices and index variables, preventing race conditions. The shared(curr, prev, width, dx2, dy2) part means that these variables are shared across all threads since they do not change during the computation and are needed by all threads.

For measuring two versions, I set one single thread for each process in the baseline version and I set one single process for many threads in the hybrid version. They are defined like this

Baseline
Number of MPI processes per node
#SBATCH --ntasks-per-node=${ntasks_per_node}

Number of openMP threads per MPI process
#SBATCH --cpus-per-task=1

Hybrid
Number of MPI processes per node
#SBATCH --ntasks-per-node=1

Varying number of openMP threads per MPI process
#SBATCH --cpus-per-task=${cpus_per_task}

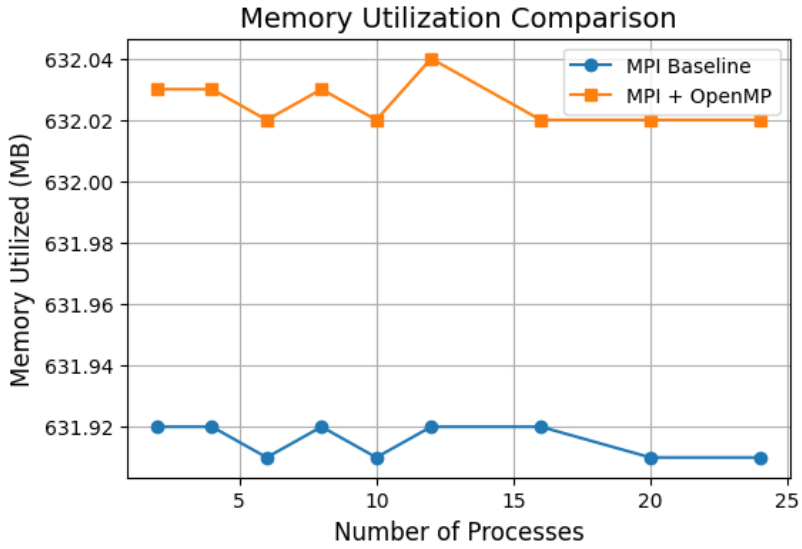
How is performance measured: I queried the running time from both versions MPI baseline and hybrid MPI by using default “time srun”. The number of tasks in baseline mode is set to equal to number of threads in hybrid mode

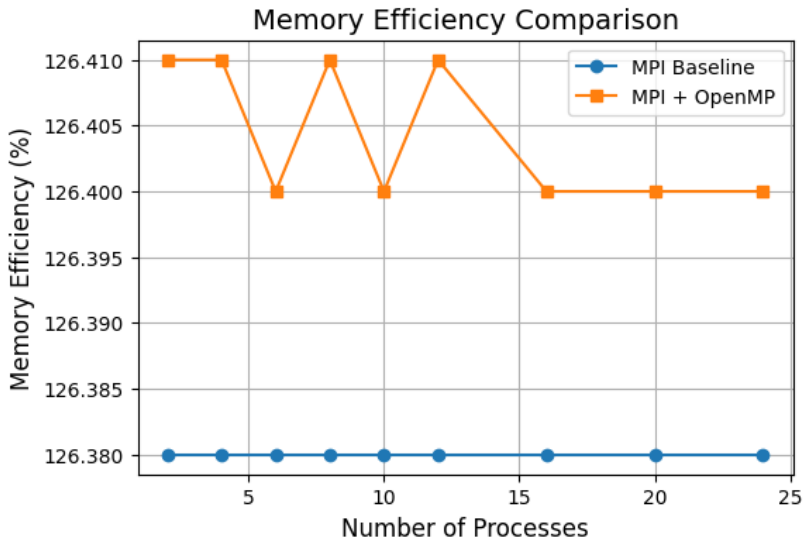
To measure memory usage, we use seff and sacct commands. The number of tasks in baseline mode is set to equal to number of threads in hybrid mode

To measure weak scaling, we vary the number of processes (= tasks x threads) and the grid size. The number of tasks in baseline mode is set to equal to number of threads in hybrid mode

To measure strong scaling, we vary only the number of processes. The number of tasks in baseline mode is set to equal to number of threads in hybrid mode

1. Reduction in memory usage





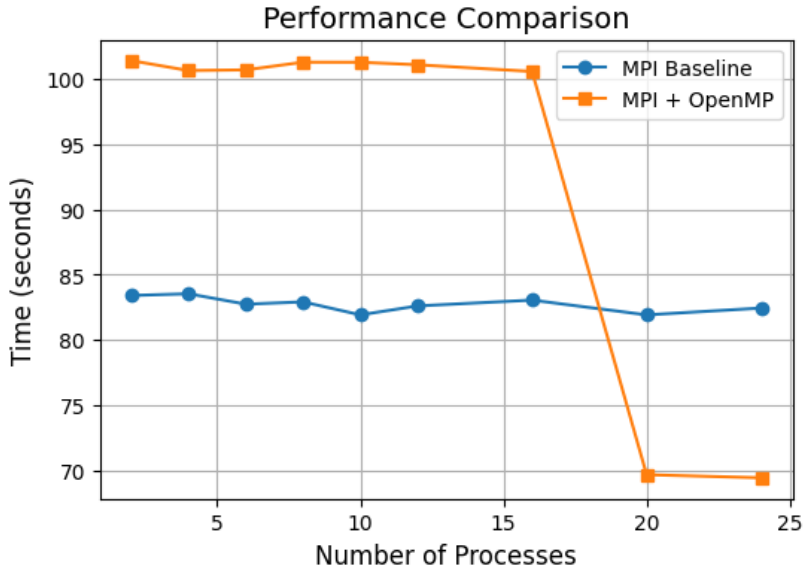
We can observe that the memory efficiency and utilization are both higher in the hybrid version compared to the baseline version. There are two possible reasons

Lower memory utilization: The MPI + OpenMP hybrid version shows higher memory utilization. This can be attributed to the efficient use of shared memory and reduced redundancy in memory allocation that OpenMP threads allow within a single MPI process.

Improved memory efficiency: With hybrid version, there is better memory efficiency due to the optimized use of cache and memory resources among the threads.

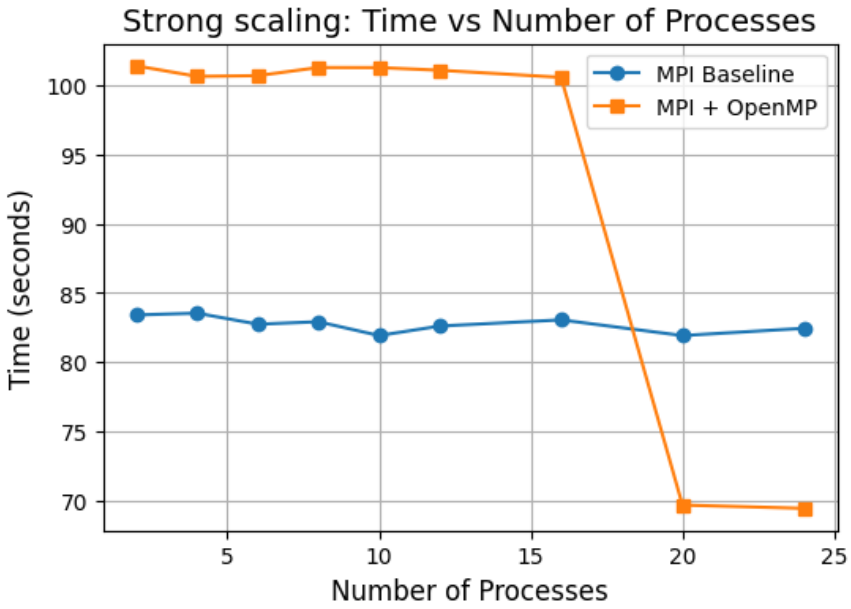
Therefore, the reduction in memory usage when we used the Hybrid version is around 0.1 MB.

2. Performance increase

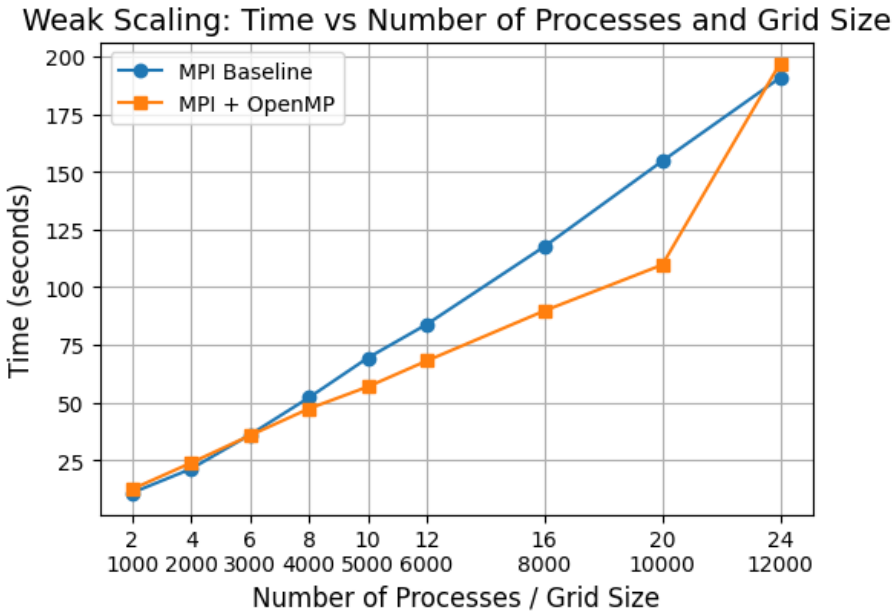


Interestingly enough, when I run the code, the Hybrid version only outperforms the Baseline version at the very high number of threads per task, which is 20 and 24 threads per task.
This could be that at lower thread counts, the overhead associated with managing multiple OpenMP threads within each MPI process in the hybrid version can outweigh the benefits of parallelism. As the number of threads per task increases (20 and 24 threads per task), this overhead becomes proportionally less significant and outweighed by the parallelism.

3. Extended scale up



This is the same graph as the performance comparison. We can see that there is no speedup at all no matter how many processes we use in the MPI Baseline version. On the other hand, the Hybrid version offers a significant improvement given enough threads. Therefore, I may conclude that the Hybrid version has better strong scaling than the baseline version



In weak scaling, the workload per processing element remains constant as the system size increases, the hybrid MPI + OpenMP approach better utilizes the available computational resources. By distributing the workload more evenly across both processes and threads, it achieves improved load balancing and computational efficiency than the baseline version