Performance Labs

Joachim Hein, LUNARC, Lund University

Acknowledgments

This tutorial is based on and utilises material developed at CSC Finland by Pekka Manninen and others, distributed under GPL.

Preface

We are going to work with a sample toy application that solves the 2D heat equation (see the Appendix in this handout) written in C. You can download that from the summer school web page.

If you prefer, you can as well work with your own code (or the code you are working with) and experiment with the techniques discussed during the lectures. You can do this in addition or instead of working with the 2D heat equation solver.

Porting and running

Unpack the tar-ball. Go to the folder mpi_p2p. This lab has been tested against the GCC 8.30 compiler. To make the gcc compiler available:

```
module swap PrgEnv-cray PrgEnv-gnu
```

Compile the code by

```
make clean

make
```

You should get an executable: `heat_mpi`. Change to the directory `../rundir_p2p`. In there you find a template submission script. Modify for the correct account and reservation. Run the code in job queue on 2, 4, 8 and 16 MPI ranks. Record the times. Are the results stable when running repeatedly?
To look at the png-file, you can use the `display` command on Tegner.

Starting the analysis with ARM-map

We now start the analysis with ARM-map

- Load the FORGE module: `module load module load allinea-forge`
- Prepare your executable in the `mpi_p2p` directory:
  - Add `-g` to the CCFLAGS of the gnu compiler section of the makefile
  - Run: `make clean`
  - Run: `make`
- Update your submission script
  - Make sure the FORGE module is loaded
  - Prefix the running of your program with: `map --prefix`
- Submit to the queue

- You should get profiles: `heat_mpi_2p_1n_2019-08-27_13-59.map`, specifying task count, node count and date.
- Analyse the profile with FORGE

When analyzing your results you will notice that the results for 16 task look quite coarse. Increase the sampling rate to achieve a better resolution. Add

```
export ALLINEA_SAMPLER_INTERVAL=2
```

to your runscript. Can you see where the problem regarding scalability is?

Assessing scalability

The application writes the current configuration every 50 steps from a single task. This inhibits scalability. Writing less frequently should improve matters. Adjust the variable `image_interval` in `main.c` to 499, to write a png-file only twice during program execution.
Rerun using 8, 16 and 32 cores. Do you see a performance improvement?
Restrict the analysis to the region between the two png-writes. How much time is spend on user code and how much time is spend in MPI? What should we aim to improve.

Node level performance

Find out in the FORGE gui whether or not the loops in the routine evolve have been vectorised or not.
To vectorise the loop, we first need to increase the optimisation level to `-O2`. At this level of optimisation vector instructions (SSE, AVX) can be issued. Rerun to see what happened. Did the code get faster? Did it vectorise?
To vectorise the working loop in the routine: `evolve` (file `core.c`), use the OpenMP simd-construct. This is portable between compilers. You need to target the correct loop. Adding the compiler flag `-fopenmp-simd` will engage the vectorisation section of OpenMP only. To get compiler feedback on loop vectorisation, add `-fopt-info-vec`. Also choose the appropriate instruction set and tuning via the `-march=...` option. Run and compare performance. Verify the vectorisation of the loop in the profiler.
Try to improve the speed of the floating-point operations. Replace the divisions in the evolve loop (in core.c) with a multiplication by pre- computed reciprocals (of the dx2 and dy2 variables). This needs to be done by hand.
Check the impact of even higher performance via the `-O3` flag. Does this help?

Optimising MPI

As the last exercise, let us see whether we can improve the MPI performance. Increase the grid size (e.g. 4096×4096) to see larger effects. Increase the iteration count, decrease the writing frequency controlled by the variable `image_interval` further and unset the `ALLINEA_SAMPLER_INTERVAL` environment variable. E.g.: to run the larger problem and with an iteration count of 4000, start the program as:

```
map --profile srun -n 32 ../mpi_p2p/heat_mpi 4096 4096 4000
```

in your jobscript.
First, we carry out an analysis with map, for the new problem size. Can you identify why does the scaling stop?

Finally, replace the blocking point-to-point communication with non-blocking operations. You can take a shortcut and use the ready implementation of the solver at the folder `mpi_ip2p`. Skim through the code and compare differences. Run it with the same core counts and compare performance. Remember to backport the changes (including the Makefile) you have made for the node-level performance optimization. Actually it might be easier to copy the `exchange` routine from the folder `mpi_ip2p` into the `core.c` file in the `mpi_p2p` folder. It is interesting to note in which MPI routine the `exchange` routine is now spending its time.

Summary

In this tutorial

- We generate performance profiles to analyse application performance
- Identify where the code is spending time
- Study load imbalance cause by conducting I/O from a single task
- We help the compiler to vectorise the key loop and study the benefit this has
- We conduct simple MPI optimisations

Even if this is a simple toy application, the tools and techniques discussed are used in real world optimisation projects. We hope you will find this useful for the future.

Appendix: 2D heat equation solver

The heat equation is a partial differential equation that describes the variation of temperature in a given region over time

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

where $u(x, y, z, t)$ represents temperature variation over space at a given time, and α is a thermal diffusivity constant. We will limit ourselves onto two dimensions (plane) and discretize the equation onto a grid. We can study the development of the temperature grid with explicit time evolution over time steps Δt:

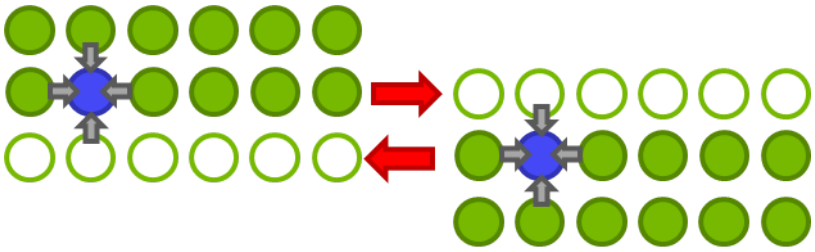
$$u^{m+1}(i, j) = u^m(i, j) + \alpha \nabla^2 u^m(i, j) \Delta t,$$

where the discretized Laplacian can be expressed as finite differences as

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

Here, Δx and Δy are the grid spacings of the temperature grid.

Parallelization of the program with MPI is pretty straightforward, the grid can be divided to tasks and the tasks can update their parts of the grid independently. The exception is the boundaries of the domains, where we need to perform the "halo exchange", i.e. before each step the last "true" column handled by a task needs to be sent to a "ghost layer" of the neighboring task.



There a solver for the 2D heat equation implemented in C in labs.tar.gz. The solver develops the equation in a user-provided grid size and over the number of time steps provided by the user. The default geometry is a disc, but user can give other shapes as input files – a bottle is provided as an example (so you can simulate how your beer will heat up when you take it to a sauna!). You can compile the program by adjusting the Makefile as needed and typing `make`. Examples on how to run the binary (remember aprun on Cray XC)

- `./heat` (no arguments - will run the program with default arguments, 2048x2048 grid and 500 time steps)
- `./heat bottle.dat` (one argument - will run the program starting from a temperature grid provided in the file bottle.dat for the default number of time steps, i.e. 500)
- `./heat bottle.dat 1000` (two arguments - will run the program starting from a temperature grid provided in the file bottle.dat for 1000 time steps)
- `./heat 1024 2048 1000` (three arguments - will run the program in a 1024x2048 rectangular grid for 1000 time steps)

The program will by default produce a .png image of the temperature field after every 50 iterations. You can change that from the parameter `image_interval` defined in `main.c`. You can view the images with some image viewer, e.g. `eog`. If the ImageMagick utility is installed, you can generate an animation of the pictures by `animate heat*.png`.