Porting and optimizing applications on Cray XC40 (March 02-04, 2015)

Workshop held at the HLRS Stuttgart, Germany

Exercise assignments

Introduction

The following exercises are to be done with your own code, as applicable. In case you do not have a code to work with, or it is unsuitable for some particular exercise, you can find two sample applications to work with, the Himeno benchmark and "Conway's Game of Life" (GoL), see the last page of this instruction set.

After logging in please change to your work directory on the lustre file system and make sure that you are work in directory distinct from all the other participants. Now copy the entire exercise material

```
> cp –r /lustre/home/esposito/KAUST_workshop/* .
```

The exercises consist of the following subdirectories:

GoL – This directory contains the C and Fortran code for "Conway's Game of Life" described in more detail below. Please leave the files in `startfiles/` unaffected and do a local copy instead. The contained Makefile targets `mpi` and `hyb` will generate a pure MPI and hybrid version of the code, respectively. The `job_mpi.pbs` and `job_hyb.pbs` are for a simple execution. More information on GoL is given below.

Himeno – Contains the C and Fortran versions of the Himeno benchmark described below. Please do a local copy of the `startfiles/` directory. The Makefile will generate per default the pure MPI version of the benchmark (only present version) which can be started via `job.pbs.` More information on the Himeno is given below.

VH1 - This code is used for the Reveal live demonstration. Reveal can assist the user in creating a hybrid program when adding OpenMP to an MPI program. It also helps to identify loop candidates for parallelization, navigate to these relevant loop candidates, view compiler optimization information, scope variables, view dependency information, and create example OpenMP directives.

The following assignment of time slots to exercises is of course not strict. Feel free to use part of a hands-on session to finish work from a previous exercise or work that is more important for your work.

Please pay attention to modify the partition and account name for the workshop in the provided job submission files

1. Porting applications [Sunday]

The aim of this first exercise is to gain first experiences with the Cray XC40 machine.

- Adapt your make files or other compilation scripts to comply with the XC environment. Remember: the compilers are always referred to as ftn, cc and CC. Build your application.
- Prepare a Moab/Torque batch job script and submit the job. Make sure it runs correctly.
- Set up a strong scaling analysis and get the corresponding strong scaling curve of your application using a representative input dataset.

In a second part of this exercise slot you are asked to recompile your application with the following flags and record the wall-clock time (or other meaningful timing information, e.g. time for one simulation step). Note that you can test these with only a few tasks using the accounts set up for the workshop. Remember to verify the output.

CCE w flags	Time	GNU w flags	Time	Intel w flags	Time	PGI w flags	Time
(no flags)		(no flags)		(no flags)		(no flags)	
-O3 -hfp3		-O3 -mavx2		-O2		-O3	
-O3 -hfp4 -Oipa4		-Ofast		-O3		-O3 -fast	
		-Ofast -funroll-loops		-Ofast		-O3 -fast -Munroll	

If the compilation time of your application is high, feel free to fill in some selected boxes only. The empty boxes are reserved for combinations of your own. Use the –craype-verbose option to the drivers to show the actually used compiler flags.

2. Performance analysis with CrayPAT [Monday Morning]

See the separate sheet “Using CrayPAT, Apprentice2 and Reveal”. You are of course free to try the performance tools directly for your application.

3. Loop statistics & performance counters [Monday Morning]

Gather data on the single-core performance of the application you are working with:

- In which loop your application spends most time?
- Investigate the following quantities for the top time-consuming routines
  - L1 and L2 cache hit ratios?
  - TLB usage?
- Run your application through the Reveal performance analysis suite. This can help to introduce OpenMP in a pure MPI application or to check an existing OpenMP implementation.

4. Single-core optimization [Monday Afternoon]

Focusing on the observed hot-spot loops and (hopefully) identified most severe problems, try to restructure the loop for better performance, with the help of the compiler feedback. Collect also other HWPC numbers for more insight. Consult us for hints! Try to identify code locations where you could restructure your code in order to use library calls instead.

5. Debugging using ATP, STAT and LGDB [Tuesday Morning]

See the separate sheet "Debugging Exercise Assignment: Using ATP, STAT, LGDB". You are of course free to try the performance tools directly for your application.

6. Improving parallel scalability [Tuesday Afternoon]

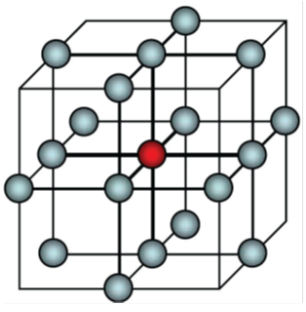
Let us try to overcome the identified scalability bottlenecks at the source code level. Please do discuss the observations and improvement strategies with us!

Alternatively or in addition, investigate the impact of the following system parameter tweaks.

- Produce again the scalability curves (strong scaling) of your application with different rank placements. Try at least the MPICH_RANK_REORDER values 0 and 2, and optionally also the optimal placement as suggested by CrayPAT.
- Experiment the impact of increasing the eager limit (MPICH_GNI_EAGER_MSG_SIZE) value.
- See the profile. If your application spends a lot of time in collective communication, try out the impact of using DMAPP collectives if you are on the XC40 system:
    - Relink your application with (e.g.) craype-hugepages8M module loaded
    - insert "export MPICH_USE_DMAPP_COLL=1" onto your batch script
    - decrease the value of MPICH_ALLTOALL_SHORT_MSG to 256

Himeno Benchmark

The code consists of a three-dimensional Poisson equation solver based on a Jacobi iteration method. An iterative loop evaluates a 19 point stencil as shown on the right hand side. The application is rather memory intensive and thus memory bandwidth bound. Fortran, C, MPI, and OpenMP implementations are available on http://accc.riken.jp/2444.htm. The present exercise set provides MPI versions for C and Fortran. With a total of ~600 source lines this benchmark is rather small but representative for a large class of simulators in science and engineering. The main quantities in this code are decomposed in domains and distributed among the processors. The information exchange between the domains is accomplished by means of a halo exchange in a non-blocking way. In the present default configuration (XL) the total grid size is 1024 x 512 x 512 and the number of iterations is fixed to 50. This is important to obtain a consistent residual value (Gosa) which serves as a validation criterion after changes to the code. In its original form, the Himeno code performs some test runs at the beginning to determine how many iterations can be done in roughly one minute. You will notice that MPI time is small in the default configuration but will become more significant when you scale to larger CPU counts. If you want to change the domain size and the processor grid you can use the paramset.sh script as follows



```
> paramset.sh <Grid size> <ID> <JD> <KD>
```

where ID x JD x KD is the size of the three-dimensional processor grid and 'Grid size' is one of the following

- XS (64x32x32)
- S (128x64x64)
- M (256x128x128)
- L (512x256x256)

- XL (1024x512x512)

For instance, the present default configuration has been obtained with

```
> paramset.sh XL 2 2 2
```

A recompilation of the code is necessary after each change of the processor grid or the computational grid. And please modify the corresponding PBS script accordingly after a change.

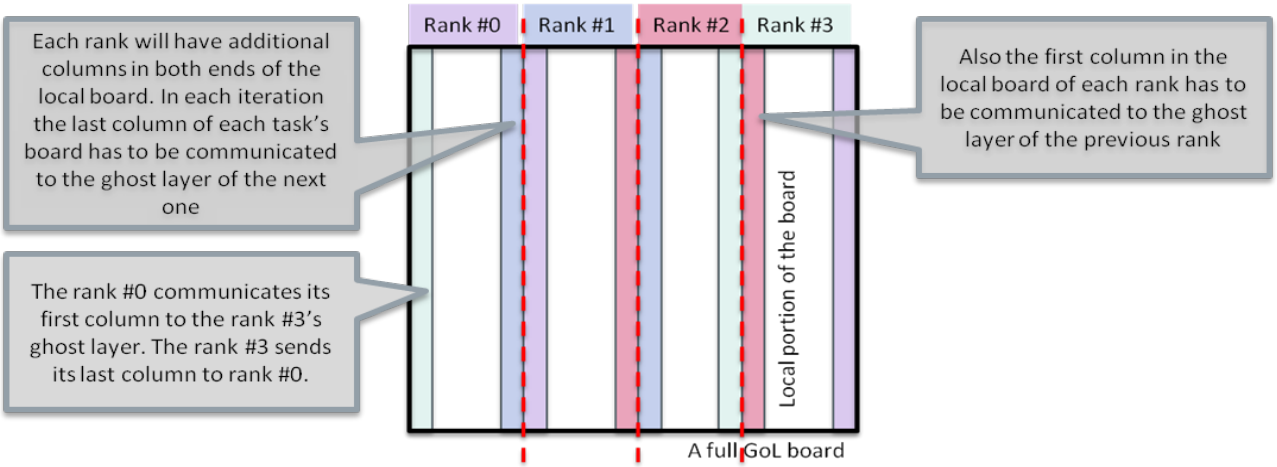
Conway's Game of Life

The *Game of Life* (GoL) is a cellular automaton devised by John Horton Conway in 1970, see **http://en.wikipedia.org/wiki/Conway's_Game_of_Life**.

Compile and run a serial reference implementation as
> make mpi
> srun –n 4 ./gol 100 4000 4000

That runs the program using a 4000x4000 board for 100 iterations. With the command **xview** you can view the images (.pbm) and see how the automaton develops.

The GoL program is parallelized with MPI, by dividing the board in columns and assigning one column to one task - a domain decomposition, that is. The tasks are able to update the board independently everywhere else than on the column boundaries - there the communication of a single column with the nearest neighbor is needed (the board is periodic, so the first column of the board is 'connected' to the last column). This is realized by having additional ghost layers on each of the local columns, which contain the boundary data of the neighboring tasks. The periodicity in the other direction is accounted for at a serial level. When printing out the board, all tasks send their local parts to one task that prints out the board.



There are, for your convenience, several rewrites of the program provided. They are

- Hybrid MPI+OpenMP version of the program (make hyb)
- Version, where the MPI_Sendrecv based halo exchange has been replaced with non-blocking communication (make nonb)
- Version, where the I/O (board image writing) has been parallelized (make pario)

It will also be straightforward to make modifications of your own into these versions.