PGAS Course Exercises

Exercise Archive

Please unpack the exercise archive by issuing the following commands

```
$ cd $HOME
$ tar -xf /lrz/sys/courses/pgas/exercises.tar
```

This will generate subdirectories **exercise_1**, **exercise_3** etc. in your home directory, containing skeleton code you can start with, as well as an **examples** subfolder. Solutions to the problems will become available successively within **/lrz/sys/courses/pgas**. Furthermore, PDF format versions of the UPC standard and extensions are available in the **/lrz/sys/courses/pgas/upc_standard** folder.

Preparation

Please choose the environment you want to use. You can use the **symmetric_data_*** program from the examples folder to check whether your setup works correctly.

To set up a SLURM batch job on the IvyMUC system used for the exercises (from a shell on lxlogin10.lrz.de with your course account), issue the following commands:

```
module load salloc_conf/ivymuc
salloc --reservation=hppa1w18_course -N 1
```

Please only reserve a single node since other participants also will need a resource. The above commands should be run **after** setting up the PGAS environments as detailed below.

Setting up the UPC environment

For UPC programming, please load the following environment:

```
module unload mkl
module load gcc/6 bupc/2.26 mkl/2018_s_gcc
```

The compilation for a static threading with 4 threads is done with

```
upcc -T4 my_upc_prog.c
```

and subsequent execution in a **salloc** shell (see section “Preparation” above) with

```
upcrun ./a.out
```

For the dynamic execution environment omit the **-T** compilation option and instead use the **-n** switch of **upcrun** to specify the number of execution threads.

Setting up the Gfortran-based CAF environment

Please load the following environment:

```
module unload mkl mpi.intel intel
module load gcc/8 mpi.intel/2018_gcc caf/gfortran mkl/2018_s_gcc
```

Compilation is done via the caf wrapper:

```
caf my_caf_prog.f90
```

and execution in a **salloc** shell via

```
cafrun –n 4 ./a.out
```

Setting up the Intel Fortran CAF environment

Please load the following environment:

```
module unload mpi.intel intel
module load intel/17.0 mpi.intel/2017 caf/intel
```

Compilation is done via the MPI wrapper:

```
mpif90 -coarray my_caf_prog.f90
```

and execution via

```
export FOR_COARRAY_NUM_IMAGES=4
./a.out
```

Exercise 1: The communication step of Matrix-Vector (45 min)

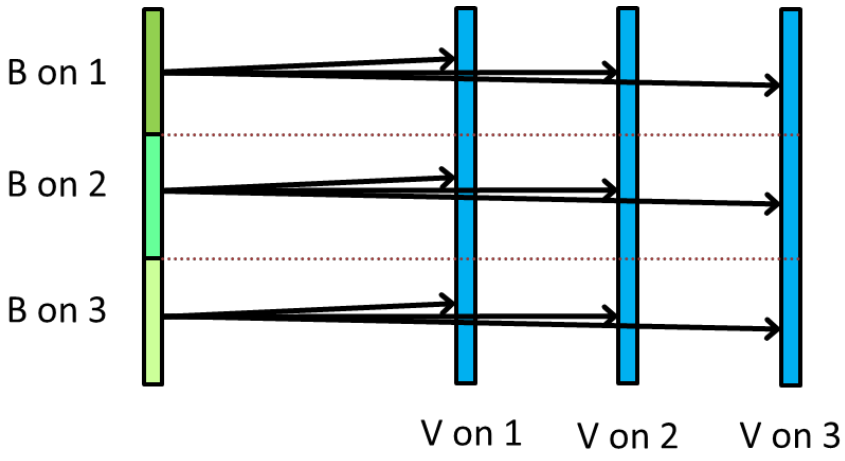
The directory **exercise_1** contains incomplete versions of the matrix-vector multiplication program discussed in the slide talk. These versions can compiled by issuing

```
make matrix_vector_upc.exe
upcrun ./matrix_vector_upc.exe
```

(the static threads execution setting is used here) or

```
make matrix_vector_caf.exe
cafrun –n 4 ./matrix_vector_caf.exe
```

but the execution will produce a run time error issued from the program because the partial result stored in array B on each task has not been propagated back to V (this is typically necessary for iterative solvers):



Implement the changes needed to do this. Which object is best selected to become a coarray/shared entity?

Note: you can modify the NEXP value supplied in the source files, but you then need to adjust the number of tasks such that (#tasks) == 4^{NEXP} . This changes the problem size such that the per-task problem size is constant ("weak scaling").

Exercise 2: Matrix-Vector with dynamic allocation (45 min)

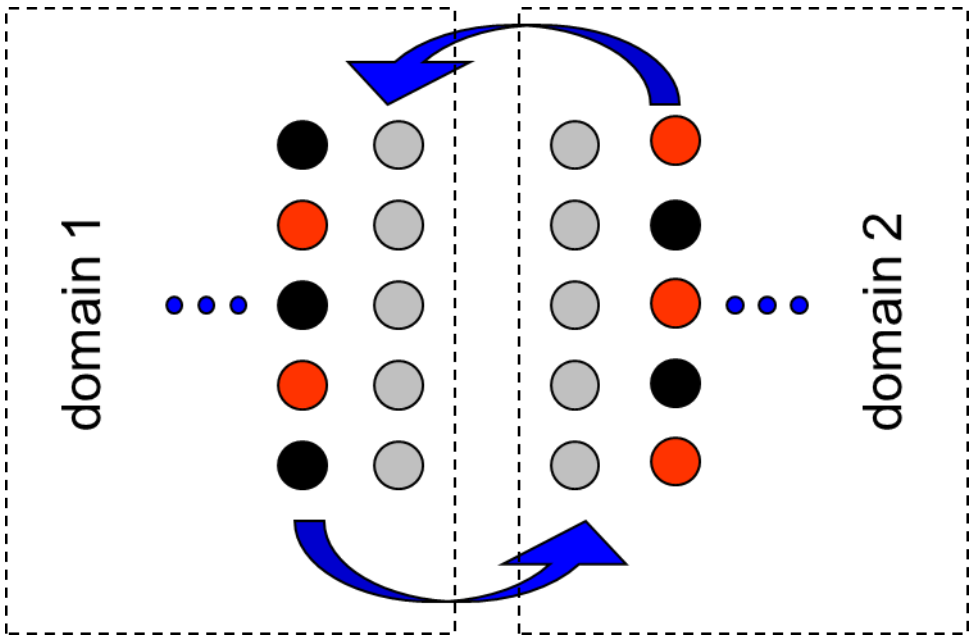
In order to make the program more flexible (e.g. by not requiring the problem size to be fixed at compile time), the required objects should be dynamically allocated. Perform the necessary changes to accomplish this. If you have not solved exercise 1, you can also start from the provided solution to it. Compile the resulting UPC program with the dynamic execution environment (i. e. without specifying the –T option).

Note: For UPC there is the additional complication that a compile-time block size MB cannot be used any more. Why is it sufficient to use a block size of 1? What do you need to take care of if this change is made?

Exercise 3: Parallelize a Jacobi solver

The serial example programs in the **`exercise_3`** subdirectory solve the heat conduction equation in two dimensions via an iterative Jacobi procedure.

a. Introduce a one-dimensional domain decomposition along the y direction. One method to deal with the boundary cell problem consists in assigning each image an additional column ("halo" or "ghost" cells, colored grey in the figure below) at its boundaries to another domain which receives data from the task that hosts that domain:



Only the halo cells need to be involved in communication; in this example, these form contiguous arrays. At first, please only run a fixed (sufficiently large) number of iterations, omitting the termination criterion.

b. Start with a small problem size to check whether correct results; the printout can be done from one (arbitrarily selected) task.
c. Implement the termination criterion.
d. When running with a problem size of 200 x 200, up to how many images does your code scale?

Exercise 4: Parallelizing a Ray Tracer

The subdirectory **`exercise_4`** contains a serial ray-tracer code (in a Fortran and a C version), which computes a pretty picture. It writes the picture to a file called 'result.pnm'. Look at the file using e.g., the 'display' program available on the front-end node. The central function is calc_tile(), which computes one tile of the picture. The size of one tile and of the whole picture is hardcoded at the start of the main program. Note that the code assumes that the picture size is a multiple of the tile size. In the version given, the picture size is 4000 x 4000 and the tile size is 200 x 200.

Parallelize the code with CAF or UPC. You can deactivate the output for testing, but make sure that your parallel code computes the correct result (this is easy since you can always display the picture). What speedup does your code obtain going from 1 to 16 tasks? Also, compare with the baseline performance from the serial code.