Some hints on the exercises for Tutorial S10:

Introduction to PGAS (UPC and CAF) and Hybrid for Multicore Programming

Supercomputing 2010

Reinhold Bader (Leibniz Supercomputing Centre) and Filip Blagojevic (Lawrence Berkeley National Laboratory)

At the beginning of the talk, please **update** the presentation PDF on your memory stick by downloading
http://portal.nersc.gov/svn/training/SC10/presentations/SCTutorial_PGAS.pdf
(quite a number of mistakes has been fixed - sorry)

The source code (serial/skeleton codes and solutions) for the exercises can be downloaded from **http://portal.nersc.gov/svn/training/SC10/**
(for example via “svn export http://portal.nersc.gov/svn/training/SC10/”)

Login to the Franklin system at NERSC

Using the test accounts handed to you by the tutorial staff, please log in to the system via secure shell:

```
ssh –X franklin.nersc.gov –l <account name>
```

If you use a Windows system, enter the host name as well as the account name into the PuTTy or windows ssh GUI.

Starting a PBS batch job

Please use the following command line to obtain dedicated cores for your tests:

```
qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:20:00 -V
```

this will give you all 4 cores of a node. If you specify mppwidth=8 (leaving all other numbers unchanged), you'll get two nodes with 4 cores for a total of 8 cores.

Using Berkeley UPC (UPC only) and running programs

```
module load bupc
```

Go to previous working directory if in a newly started PBS shell:

```
cd $PBS_O_WORKDIR
```

Compilation:

```
upcc  -O  -o myprog myprog.c
```

Parallel Execution (requires a PBS shell):

```
upcrun -n 1 -cpus-per-node 4 ./myprog
upcrun -n 2 -cpus-per-node 4 ./myprog
upcrun -n 4 -cpus-per-node 4 ./myprog
```

Note that the standard environment should be used (PrgEnv-pgi) to make the above "module load" work correctly.

Using the Cray compilers (UPC and CAF) and running programs

```
module switch PrgEnv-pgi PrgEnv-cray
```

Go to previous working directory if in a newly started PBS shell:

```
cd $PBS_O_WORKDIR
```

UPC Compilation:

```
cc -h upc -o myprog myprog.c
```

or CAF compilation:

```
ftn -e m -h caf -o myprog myprog.f90
```

Parallel Execution (requires a PBS shell):

```
aprun -n 1 -N 1 ./myprog
aprun -n 2 -N 2 ./myprog
aprun -n 4 -N 4 ./myprog
```

Login and Usage of the backup system at LRZ (Munich)

This system will only be used for the exercises in case the NERSC system “Franklin” has a failure. Otherwise, this section can be ignored.
The backup system at LRZ in Munich is an SGI Ultraviolet with 256 cores. To log in to the system, please issue the command

```
ssh –X lx64ia2.lrz.de –l <account name>
```

(these accounts have **different** names from those on Franklin and will only be handed out if the emergency case occurs) and from this front end node type

```
ssh –X uv1
```

making use of the backup account names and the password provided by the tutorial staff. The software environment for UPC consists of a **Berkeley UPC** installation:

```
module load upc/bupc
upcc [-g –T=<number of threads>] –o myprog.exe myprog.upc
upcrun [–n <number of threads>] ./myprog.exe
```

If the number of threads is fixed at compilation time, a run time specification of thread number must be consistent.
A **coarray Fortran** prerelease of Intel’s newest 12.0 compiler (configured for shared memory operation) is also available on the Ultraviolet:

```
module load intel_caf          #  (please ignore any error messages appearing here)
ifort –coarray –o mycafprog.exe mycafprog.f90
export FOR_COARRAY_NUM_IMAGES=<number of images>
./mycafprog.exe
```

Please do not use more than a few (4-6) images for executing your programs. The modules for coarray Fortran and Berkeley UPC should not be loaded at the same time within one shell.

Hints for Exercise 1 (Presentation slide 40)

This exercise consists of **two parts**.
The **first part** of the exercise is concerned with

- successfully compiling and running the most simple possible CAF / UPC program with more than one image / thread
- printing information about image / thread identity from inside the program.

The solution for this first part of the exercise is available in the presentation on slide 136.
The **second part** of the exercise asks you to add

- an additional declaration of a coarray / UPC shared object
- some additional code to initialize the object on one image, and write out the value of the initialized object on another image

and run the resulting executable with **at least two** images / threads. Due to missing synchronization statements, inconsistent results might (or might not) be obtained – the program is **not** conforming.

Part 1 of Exercise 1

Please make use of the **`THIS_IMAGE()`** as well as **`NUM_IMAGES()`** intrinsic functions in the case of CAF, and the **`THREADS`** and **`MYTHREAD`** function macros in case of UPC. The meaning of these functions is as follows:

Coarray Fortran	UPC	explanation
`NUM_IMAGES()`	**`THREADS`**	default integer function / int function macro which returns the number of images / threads with which the program is executed
`THIS_IMAGE()`	**`MYTHREAD`**	default integer function / int function macro returning a unique value • between 1 and NUM_IMAGES() in Fortran • between 0 and THREADS – 1 in UPC depending on which image / thread executes the statement.

The solution can also be found in **`hello/solutions/hello_part1.upc`** or **`hello/solutions/hello_part1.f90`**

Part 2 of Exercise 1

Declarations of **coarrays** in **Fortran** must (like other declarations) happen **before** the first executable statement. A scalar coarray declared as

```
integer :: x[*]
```

can then be defined locally via an executable statement like

```
x = this_image()
```

resulting in an image dependent value being stored, and the value stored on image 1 might later be referenced from a remote image using a statement like

```
if (this_image() > 1) write(*,*) 'x from 1 is: ',x[1]
```

The number in angular brackets is called the coindex; mapping between image index (i.e. the result of **`THIS_IMAGE()`**) and coindex is trivial here, but may be more complicated in general.

Declarations of **shared entities** in **UPC** must often be performed outside function bodies, especially if a dynamic THREADS environment is provided. The reason for this is that many kinds of shared entities are not allowed to be automatic. A declaration essentially equivalent to the above coarray declaration with respect to how data are distributed among tasks may for example be

```
shared [1] int x[THREADS];
```

and inside the main() function a local definition equivalent to the coarray case given above might be

```
  x[MYTHREAD] = MYTHREAD + 1;
```

and the subsequent reference to remote data then is

```
   if (MYTHREAD > 0) printf("x from 0 is: %i\n",x[0]);
```

Note that, in contrast to CAF, array notation is used and the normal array index is mapped to the thread index. This mapping may be more complicated in general than in this simple example.

Running the executable will occasionally result in inconsistent values printed from different threads. The reason for this is that your code is not valid since a synchronization statement is required between definition by one task and reference by another one. **Corrected** solutions may be found in **`hello/solutions/hello_part2_CORRECTED.upc`** and **`hello/solutions/hello_part2_CORRECTED.f90`**.

Hints for Exercise 2 (Triangular matrix, presentation slides 75-76)

Reading in the problem parameters from standard input

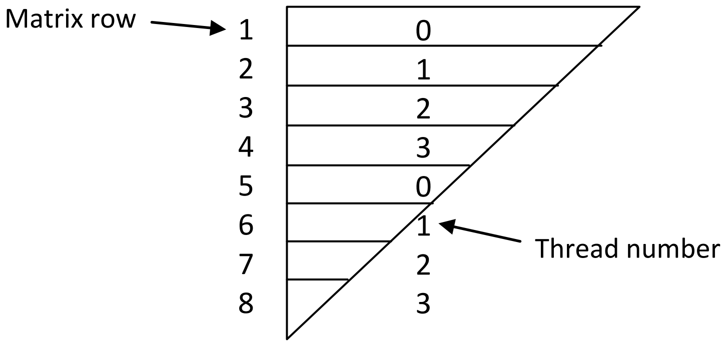
This should be done only on image 1 (CAF) or thread 0 (UPC). As a consequence, some method must be implemented to propagate the value to all other images, involving synchronization statements. In UPC, providing shared entities as arguments to library functions (in this exercise, `scanf()`) requires suitable casting.

Type definitions and allocations

In order to achieve balanced usage of memory, a **cyclic** data distribution is recommended: For example, in the case of four tasks the mapping between row index and tasks is given by

Thread (UPC)	0	1	2	3
Image (CAF)	1	2	3	4
Row index	1,5,9,13,17,...	2,6,10,14,18,...	3,7,11,15,19,...	4,8,12,16,20,...

For the case of UPC threads (THREADS==4), the reverse mapping is also indicated graphically:



Since the number of rows is determined at run time, it will be necessary to declare a dynamic entity:

CAF	UPC
type(tri_matrix), allocatable :: a(:)[:]	shared [ ] Tri_matrix * shared[1] A[THREADS];

For this example, it would not be necessary to use shared data, but this would change if further matrix operations (e.g., matrix multiplication) must be implemented which require cross-task data access. Each task will contain the same number of rows. Allocation then is done via

CAF	UPC
allocate(a(rows_per_proc)[*])	A[MYTHREAD] = (shared [ ] Tri_matrix *) upc_alloc(rows_per_thread * sizeof(Tri_matrix));

Here we use the **non-collective** call `upc_alloc()` to assign each pointer array element its own i.e., **thread-individual** shared area with affinity to the executing thread. This is in contrast to using `upc_all_alloc()` which is **collective** and returns the **same** shared entity to each pointer, or `upc_global_alloc()`, which is also non-collective, but returns a distributed shared pointer.
In UPC, it will be necessary to change the type definition as follows to avoid the spurious situation of generating shared pointers to local entities:

```
typedef struct {
  shared [] float *row;
```

```
   size_t row_size;
} Tri_matrix;
```

The block size of zero implies that the pointer will point to a piece of memory which is shared, but with affinity to only one thread. The allocation must hence be performed using **`upc_alloc()`**.

Loop structure for cyclic processing

This is provided on slide 52 of the presentation. Slide 137 gives part of the coarray Fortran solution as well as the location of the complete solution program sources (both CAF and UPC).

Comments on the alternative exercise

The first step for this exercise is to understand the solution program provided in triangular_matrix/solutions/triangular.[f90|upc].

Then, you need to make the following changes:

- an additional synchronization statement is needed since printing from all tasks involves cross-image / cross-thread references
- when removing the IF statement, one must take care that access to the local data in the WRITE/printf statement must be converted to an access to shared data.
    - In CAF be aware that a(…) is a local reference and must be substituted by a(…)[…]
    - In UPC the local reference via MYTHREAD must be appropriately changed
- In the case of CAF, unfortunately the program will not work when using the Cray or Intel compilers (there appear to be compiler bugs with respect to referencing of type components).

Hints for Exercise 3 (Reduction, presentation slide 109)

This example is targeted at Fortran programmers (since UPC provides collectives out of the box). Skeleton code to start off with is contained in **`reduction_heat/mod_reduction.f90`**, and a simple test program is in **`reduction_heat/test_reduction.f90`**. It is recommended to

- declare a global coarray in the module's specification part for intermediate results
- use a critical region for implementing the reduction
- make use of the information on the updated slide 78 for the prefix reduction

Solutions and pointers to solution source code can be found on slides 138 and 139.

Note: The Cray compiler unfortunately has a bug which causes wrong results and/or crashes for procedure invocations with coindexed actual arguments. To work around this, please replace a line like

```
call foo(x[5])
```

by

```
y = x[5]
call foo(y)
x[5] = y        ! if dummy is modified
```

Hints for Exercise 4 (Presentation slide 110)

Coarray Fortran version

The idea here is to perform a one-dimensional block decomposition with halo data exchange as sketched on slide 98 of the presentation. A serial version of the code is provided as

`reduction_heat/heat_serial.f90`

To keep things easy, the image-local block size **`kmax_loc`** is chosen to be the same on each image such that the product **`kmax_loc * num_images()`** is as close as possible to **`kmax`**. To be flexible, it is recommended to give the coarray field the **`ALLOCATABLE`** attribute.

The iteration process is controlled by the global difference of the temperature fields between iterations (variable **`dphimax`**); it is therefore necessary to perform a reduction here, for which you should use the module **`mod_reduction.f90`** developed in exercise 3. A simple example of how to use it in this context is provided as

`reduction_heat/test_reduction.f90`

For small problem sizes (**`imax == 20`**, **`kmax == 11`**), a printout of the temperature field is generated before the first iteration step is performed, and after convergence has been reached. Please run the serial version of program with test output to see what the results should look like. Note that due to the slight changes in discretization length along the parallelized dimension the numbers may be slightly different depending on the number of images used, but at the end of the iteration process **all numbers within a column** should be the same. For the parallel version, it is for the purpose of this exercise considered sufficient to perform a partial printout from one (arbitrarily selected) image.

Some variants of solutions are available as

`reduction_heat/solutions/heat_halo.f90`
`reduction_heat/solutions/heat_halo_push.f90`
`reduction_heat/solutions/heat_nohalo.f90` (does not use coarrays for full fields)

Note on performance: On x86 based systems, no speedup (but rather a slowdown) is observed for problem size 100 x 100. Much bigger problem sizes may be better (but take a long time to complete). Coarray implementations are not mature enough yet to properly optimize tightly coupled codes such as this one.

UPC version

Note on Cray compilers: We detected certain memory allocation problems with this example when Cray UPC is used; to avoid possible problems, we suggest compiling the code with a fixed thread number using the –X option e.g.:

```
cc -h upc -X 4 -o ./heat.exe heat.upc
```

The idea here is to use a one-dimensional block decomposition and memory block transfer functions as sketched on slide 100 of the updated presentation. A serial version of the code is provided as

`reduction_heat/heat_serial.c`

To keep things easy, the thread-local block size **`imax_loc`** is chosen to be the same on each image such that the product **`imax_loc * THREADS`** is as close as possible to **`imax`**. Suitable shared arrays for the block transfer can be declared as

```
shared [KMAX] double phi_top[THREADS][KMAX];
shared [KMAX] double phi_bot[THREADS][KMAX];
```

The iteration process is controlled by the global difference of the temperature fields between iterations (variable **`dphimax`**); it is therefore necessary to perform a reduction here, for which you should use the reduction function **`upc_all_reduceD()`** which is described on slide 95 of the presentation; **`UPC_MAX`** must be used for the operation argument. A simple example of how to use it in this context is provided as

`reduction_heat/solutions/reduction.upc`

For small problem sizes (**`imax == 20`**, **`kmax == 11`**), a printout of the temperature field is generated before the first iteration step is performed, and after convergence has been reached. Please run the serial version of program with test output to see what the results should look like. Note that due to the slight changes in discretization length along the parallelized dimension the numbers may be slightly different depending on the number of images used, but at the end of the iteration process **all numbers within a column** should be the same. For the parallel version, it is for the purpose of this exercise considered sufficient to perform a partial printout from one (arbitrarily selected) thread.

Some variants of solutions are available as

```
reduction_heat/solutions/heat_nohalo.upc
reduction_heat/solutions/heat_halo.upc
```

Note on performance: On x86 based systems, reasonably good scaling up to 4 threads, but relatively bad base line (1 thread) performance (when compared to the gcc-compiled serial version) is observed for problem size 100 x 100. For tightly coupled applications such as this one UPC implementations still have room for improvement.

Distributed structure example (Presentation slides 106-108)

Illustrative code for this example is available as the following files:

`distributed_structures/data.c`	
`distributed_structures/data.h`	
`distributed_structures/test_tree.upc`	(main program)
`distributed_structures/tree.h`	(prototypes)
`distributed_structures/tree.upc`	(tree manipulations)

for you to inspect and run. Note that a destructor has not been implemented. This example requires the use of Berkeley UPC (the executable produced by the Cray compiler crashes).

MPI-UPC Hybrid Example

PGAS languages provide shared global address space that may include memory from a large number of distinct physical nodes. Large amount of memory (not necessarily physically contiguous) allow PGAS applications to work with extremely large datasets. To exploit the benefits of the PGAS programming models, programmers are often required to rewrite existing MPI parallel code, using some of the PGAS languages. Hybrid MPI-UPC programming model emerges as an alternative to the arduous application-rewriting task. The hybrid model increases the amount of memory available to a single MPI task, but also increases the scalability of certain UPC applications. James Dinan et al. (*) observed significant performance benefit when a random access benchmark and the Barnes-Hut cosmological simulations are parallelized using the MPI-UPC hybrid programming model.

The MPI-UPC hybrid programming concepts are still in the prototyping phase, and here we provide only a short introduction. More detailed information about the MPI-UPC hybrid programming models can be found on the websites:

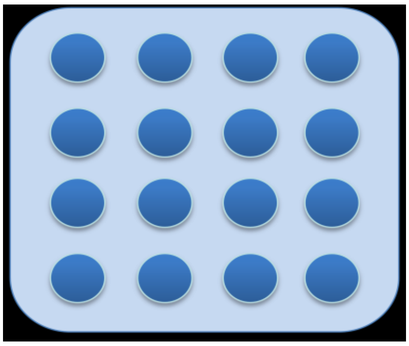
http://barista.cse.ohio-state.edu/wiki/index.php/Hybrid_MPI%2BUPC
http://upc.lbl.gov/docs/user/interoperability.shtml#mpi

and in the paper:

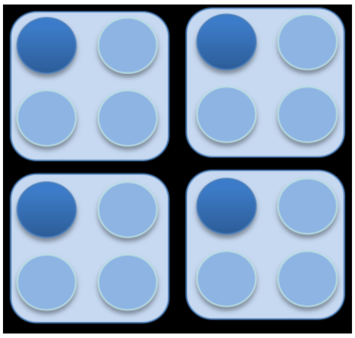
(*) J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, “Hybrid Parallel Programming with MPI and Unified Parallel C,” *7th ACM International Conference on Computing Frontiers*, ACM, Bertinoro, Italy, 2010.

Three basic MPI-UPC programming models can be recognized:

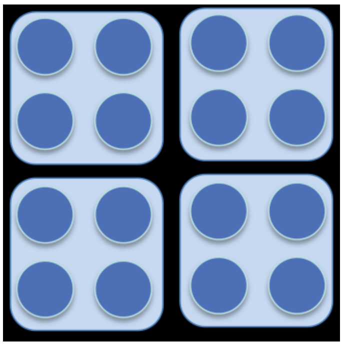
1) Flat – All processes participate in both MPI and UPC communication
2) Nested-Funneled – Multiple groups of UPC processes exist, and only a single process from each group performs MPI communication. UPC processes can communicate only within the group. Note that each UPC group can span multiple nodes.
3) Nested-Multiple – Multiple groups of UPC processes exist, and MPI spans all processes (all processes can participate in MPI communication). UPC processes can communicate only within the group, and a UPC group can span multiple nodes



Flat Model



Nested-Funneled Model



Nested-Multiple Model

Running the MPI-UPC hybrid code:

- Flat model requires the Berkeley UPC runtime to be configured with CC, MPI_CC and CXX set to MPI C and C++ compilers. Also, when compiling the application passing --uses-mpi to upcc is required (see http://upc.lbl.gov/docs/user/interoperability.shtml#mpi). With ”--uses-mpi”, the Berkeley UPC runtime will call MPI_Init() and MPI_Finalize() functions, and therefore these functions need to be removed from the application.

- For nested models, the Berkeley UPC Runtime needs to be configured with the SSH spawner and the --disable-mpi option. For example, on an InfiniBand platform the following options must be passed to the configure script: --with-ibv-spawner=ssh --with-vapi-spawner=ssh --disable-mpi
- Running commands – while the flat model can be spawned with upcrun, the nested models require the process manager with MPMD support (for example the Hydra process manager which is available with mpich2 installation):
  - Flat: upcrun -n N ./app (app is previously compiled using “--uses-mpi”)
  - Nested-Funneled, MPMD launching support is required:
    ```
    mpiexec.hydra -f  <HOSTFILE>
    -env GASNET_SSH_SERVERS=<HOSTS> upcrun -n N ./app :
    -env GASNET_SSH_SERVERS=<HOSTS> upcrun -n N ./app
    ```
  - Nested-Multiple:
    ```
    mpiexec.hydra -ranks-per-proc=N -f  <HOSTFILE>
    -env GASNET_SSH_SERVERS=<HOSTS> upcrun -n N ./app :
    -env GASNET_SSH_SERVERS=<HOSTS> upcrun -n N ./app
    ```

The control of the UPC process distribution needs to be done through GASNET_SSH_SERVERS environment variable, since the SSH spawner is used.

Recommendation: start with the Flat (simplest) model!

While in the flat and nested-multiple models all processes are involved in the MPI communication, in the nested-funneled model, only one process is allowed to call MPI functions.

Nested-Funneled model example:

```
#include <stdio.h>
#include <upc.h>
#include <mpi.h>

#define DSOLOCAL
int main (int argc, char *argv[])
{
  int rank, size;

  if (MYTHREAD==0){
    MPI_Init (&argc, &argv);    /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);  /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
    printf( "Hello world from UPC process: %d of %d, MPI process %d of %d\n", MYTHREAD,
                    THREADS, rank, size );
    MPI_Finalize();
  } else {
    printf( "Hello world from upc process %d of %d\n", MYTHREAD, THREADS );
  }
  return 0;
}
```