Programming Models and Languages for Clusters of Multi-core Nodes

Part 3: PGAS Languages

Alice Koniges – NERSC, Lawrence Berkeley National Laboratory

Rolf Rabenseifner – High Performance Computing Center Stuttgart (HLRS), Germany

Gabriele Jost – Texas Advanced Computing Center, The University of Texas at Austin

*Georg Hager – Erlangen Regional Computing Center (RRZE), University of Erlangen-Nuremberg, Germany

*author only—not speaking

Tutorial at SciDAC Tutorial Day
June 19, 2009, San Diego, CA

Outline

- **Introduction**
  - PGAS languages and comparison with MPI and OpenMP
- **Let's start**
  - The process model → Practical 1 (hello)
- **PGAS Data and Barrier Synchronization**
  - Data distribution and remote data access → Practical 2 (neighbor)
- **Work-Distribution and Other Synchronization Concepts**
  - Lock synchronization, critical sections, forall, … → Practical 3 (pi)
- **Arrays and Pointers**
  - Dynamic allocation, multi-dimensional arrays, … → [Practical 4 (heat)]
- **Wrap up & Summary**
- **Annex**

https://fs.hlrs.de/projects/rabenseifner/publ/SciDAC2009-Part3-PGAS.pdf
(the pdf includes additional "skipped" slides)

Acknowledgements

- Andrew A. Johnson — for his first ideas about teaching UPC and CAF
- Gabriele Jost — for PGAS benchmark slides
- Alice Koniges — who organized a first PGAS tutorial at SciDAC2009
- Ewing (Rusty) Lusk — who gave me additional code examples
- Tünde Erdei and
- Oliver Mangold — for their PGAS support at HLRS
- All other — who set up the environment on several platforms

https://fs.hlrs.de/projects/rabenseifner/publ/SciDAC2009-Part3-PGAS.pdf

Introduction

https://fs.hlrs.de/projects/rabenseifner/publ/SciDAC2009-Part3-PGAS.pdf

Partitioned Global Address Space (PGAS) Languages

- **Co-Array Fortran (CAF)**
- **Unified Parallel C (UPC)**
- **Titanium (Java based)**

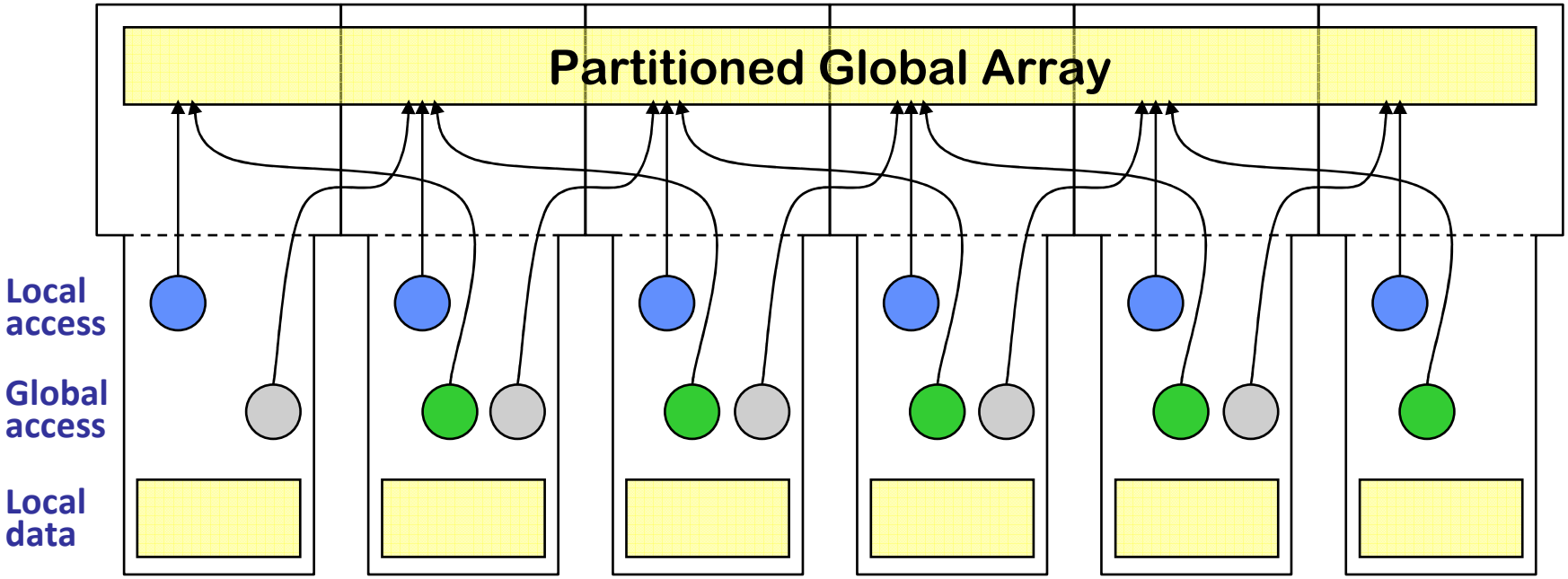
DARPA High Productivity Computer Systems (HPCS) language project:

- **X10 (based on Java, IBM)**
- **Chapel (Cray)**
- **Fortress (SUN)**

https://fs.hlrs.de/projects/rabenseifner/publ/SciDAC2009-Part3-PGAS.pdf

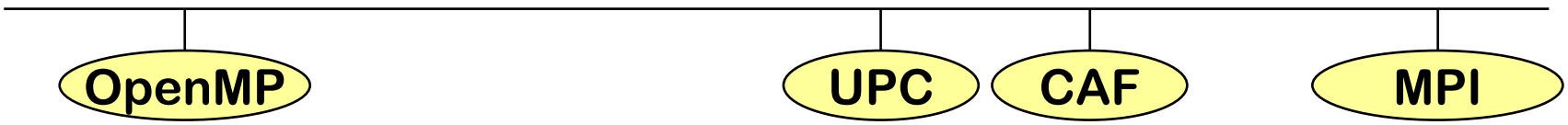
Partitioned Global Address Space (PGAS) Languages

- **Principles:**



Other Programming Models

- **Message Passing Interface (MPI)**
  - Library with message passing routines
- **OpenMP**
  - Language extensions with shared memory worksharing directives



- **UPC / CAF data accesses:**
  - Similar to OpenMP
- **UPC / CAF worksharing:**
  - Similar to MPI

Support

- **PGAS in general**
  - http://en.wikipedia.org/wiki/PGAS
  - http://www.pgas-forum.org/ → PGAS conferences
- **UPC**
  - http://en.wikipedia.org/wiki/Unified_Parallel_C
  - http://upc.gwu.edu/ → Main UPC homepage
  - https://upc-wiki.lbl.gov/UPC/ → UPC wiki
  - http://upc.gwu.edu/documentation.html → Language specs
  - http://upc.gwu.edu/download.html → UPC compilers
- **CAF**
  - http://en.wikipedia.org/wiki/Co-array_Fortran
  - http://www.co-array.org/ → Main CAF homepage
  - Part of upcoming Fortran 2008
  - http://www.g95.org/coarray.shtml → g95 compiler

skipped

UPC

- The UPC Consortium:
  UPC Language Specification (V 1.2)
  June 2005
  - http://upc.gwu.edu/docs/upc_specs_1.2.pdf

- Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, Tarek El-Ghazawi:
  UPC Manual
  May 2005
  - http://upc.gwu.edu/downloads/Manual-1.2.pdf

skipped

CAF

- From http://www.nag.co.uk/SC22WG5/
- John Reid:
  Co-arrays in the next Fortran Standard
  ISO/IEC JTC1/SC22/WG5 N1772 (2009)
  - ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800/N1772.pdf

Older versions:

- Robert W. Numrich and John Reid:
  Co-arrays in the next Fortran Standard
  ACM Fortran Forum (2005), 24, 2, 2-24 and WG5 paper ISO/IEC JTC1/SC22/WG5 N1642
  - ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1642.pdf
- Robert W. Numrich and John Reid:
  Co-Array Fortran for parallel programming.
  ACM Fortran Forum (1998), 17, 2 (Special Report) and Rutherford Appleton Laboratory report RAL-TR-1998-060 available as
  - ftp://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf

Parallelization strategies — hardware resources

- Two major resources of computation:
  - **processor**
  - **memory**
- Parallelization means
  - **distributing work to processors**
  - **distributing data (if memory is distributed)**

  and
  - **synchronization of the distributed work**
  - **communication of *remote* data to *local* processor** (if memory is distributed)
- Programming models offer a combined method for
  - distribution of work & data, synchronization and communication

skipped

Distributing Work & Data

Work decomposition

- based on loop decomposition

do i=1,100
➔ i=1,25
i=26,50
i=51,75
i=76,100

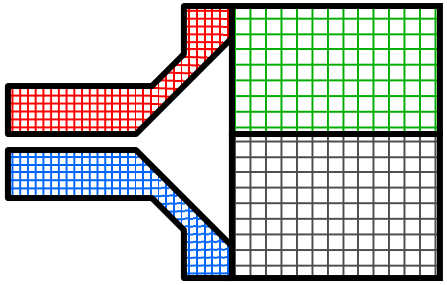
Data decomposition

- all work for a local portion of the data is done by the local processor

A(1:20, 1: 50)
A(1:20, 51:100)
A(21:40, 1: 50)
A(21:40, 51:100)

Domain decomposition

- decomposition of work and data is done in a higher model, e.g. in the reality



skipped

Synchronization

```
Do i=1,100
    a(i) = b(i)+c(i)
Enddo
Do i=1,100
    d(i) = 2*a(101-i)
Enddo
```

i=1..25 | 26..50 | 51..75 | 76..100
execute on the 4 processors

BARRIER synchronization

i=1..25 | 26..50 | 51..75 | 76..100
execute on the 4 processors

- Synchronization
  - is necessary
  - may cause
    - idle time on some processors
    - overhead to execute the synchronization primitive

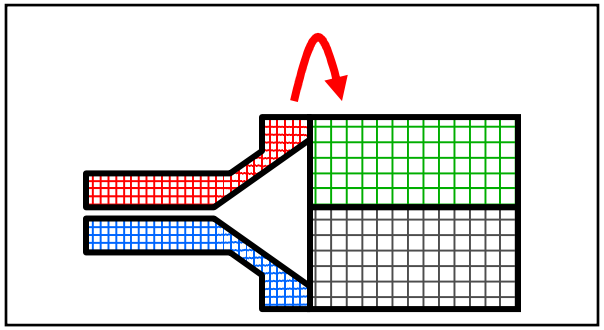
skipped

Communication

```
Do i=2,99
    b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

- Communication **is necessary on the boundaries**
  - e.g. b(26) = a(26) + f*(a(25)+a(27)-2*a(26))
  - e.g. at domain boundaries

a(1:25),	b(1:25)
a(26,50),	b(51,50)
a(51,75),	b(51,75)
a(76,100),	b(76,100)



skipped

Major Programming Models

1 OpenMP
- Shared Memory **Directives**
- to define the work decomposition
- no data decomposition
- synchronization is implicit (can be also user-defined)

- HPF (High Performance Fortran)
  - Data Parallelism
  - User specifies data decomposition with **directives**
  - Communication (and synchronization) is implicit

- MPI (Message Passing Interface)
  - User specifies how work & data is distributed
  - User specifies how and when communication has to be done
  - by calling MPI communication **library-routines**

skipped

Shared Memory Directives – OpenMP, I.

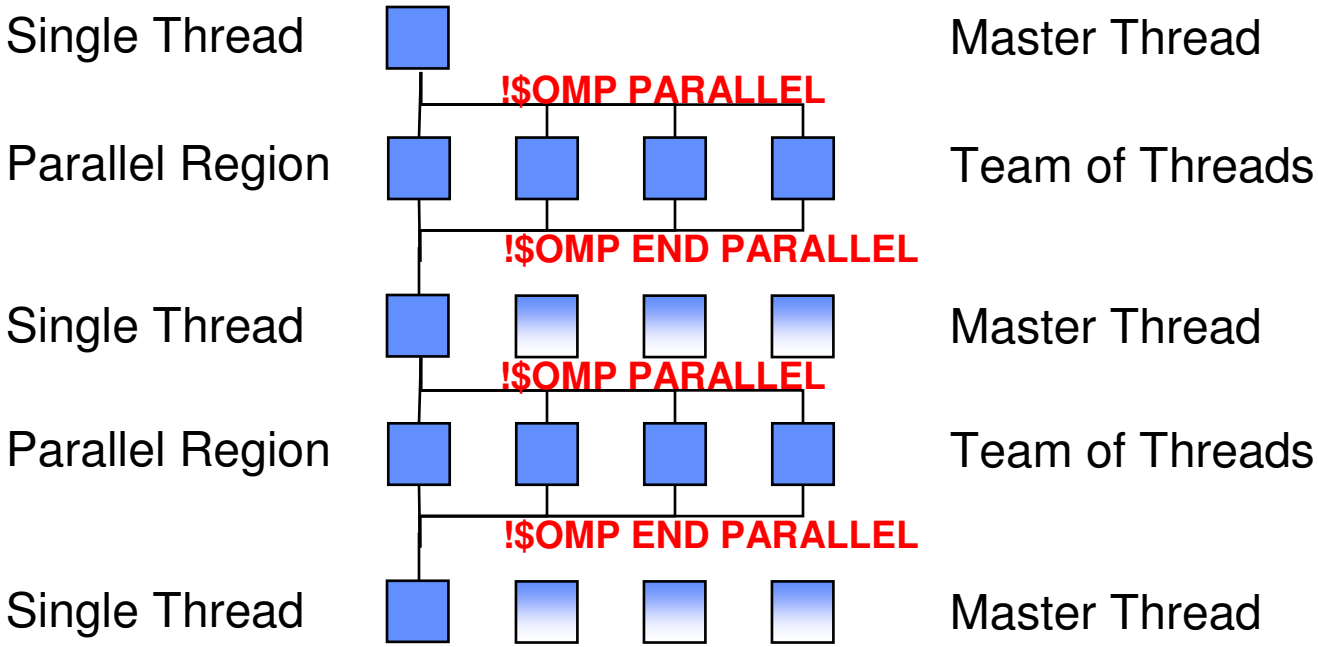
OpenMP

```
Real :: A(n,m), B(n,m)

!$OMP PARALLEL DO
do j = 2, m-1
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
!$OMP END PARALLEL DO
```

- Data definition
- Loop over y-dimension
- Vectorizable loop over x-dimension
- Calculate B, using upper and lower, left and right value of A

skipped

Shared Memory Directives – OpenMP, II.



skipped

Shared Memory Directives – OpenMP, III.

- OpenMP
  - standardized shared memory parallelism
  - thread-based
  - the user has to specify the work distribution explicitly with directives
  - no data distribution, no communication
  - mainly loops can be parallelized
  - compiler translates OpenMP directives into thread-handling
  - standardized since 1997
- Automatic SMP-Parallelization
  - e.g., Compas (Hitachi), Autotasking (NEC)
  - thread based shared memory parallelism
  - with directives (similar programming model as with OpenMP)
  - supports automatic parallelization of loops
  - similar to automatic vectorization

skipped

Major Programming Models – HPF

HPF

1. OpenMP
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. HPF (High Performance Fortran)
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (and synchronization) is implicit

- MPI (Message Passing Interface)
  - User specifies how work & data is distributed
  - User specifies how and when communication has to be done
  - by calling MPI communication **library-routines**

skipped

Data Parallelism – HPF, I.

```
Real :: A(n,m), B(n,m)
!HPF$ DISTRIBUTE A(block,block), B(...)

do j = 2, m-1
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```

- Data definition
- Loop over y-dimension
- Vectorizable loop over x-dimension
- Calculate B,
  - using upper and lower,
    - left and right value of A

skipped

Data Parallelism – HPF, II.

- **HPF (High Performance Fortran)**
  - standardized data distribution model
  - the user has to specify the data distribution explicitly
  - Fortran with language extensions and directives
  - compiler generates message passing or shared memory parallel code
  - work distribution & communication is implicit
  - set-compute-rule:
    the owner of the left-hand-side object computes the right-hand-side
  - typically arrays and vectors are distributed
  - draft HPF-1 in 1993, standardized since 1996 (HPF-2)
  - JaHPF since 1999

skipped

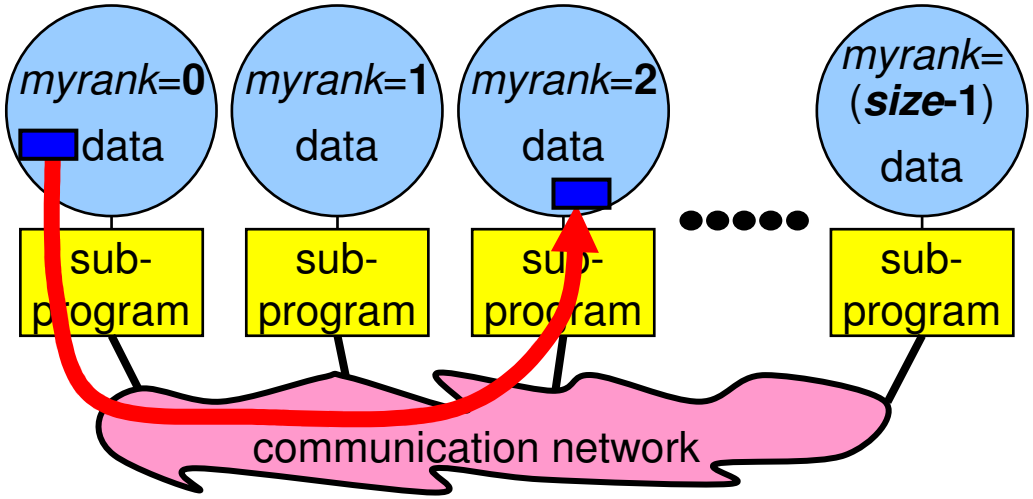
Major Programming Models – MPI

MPI

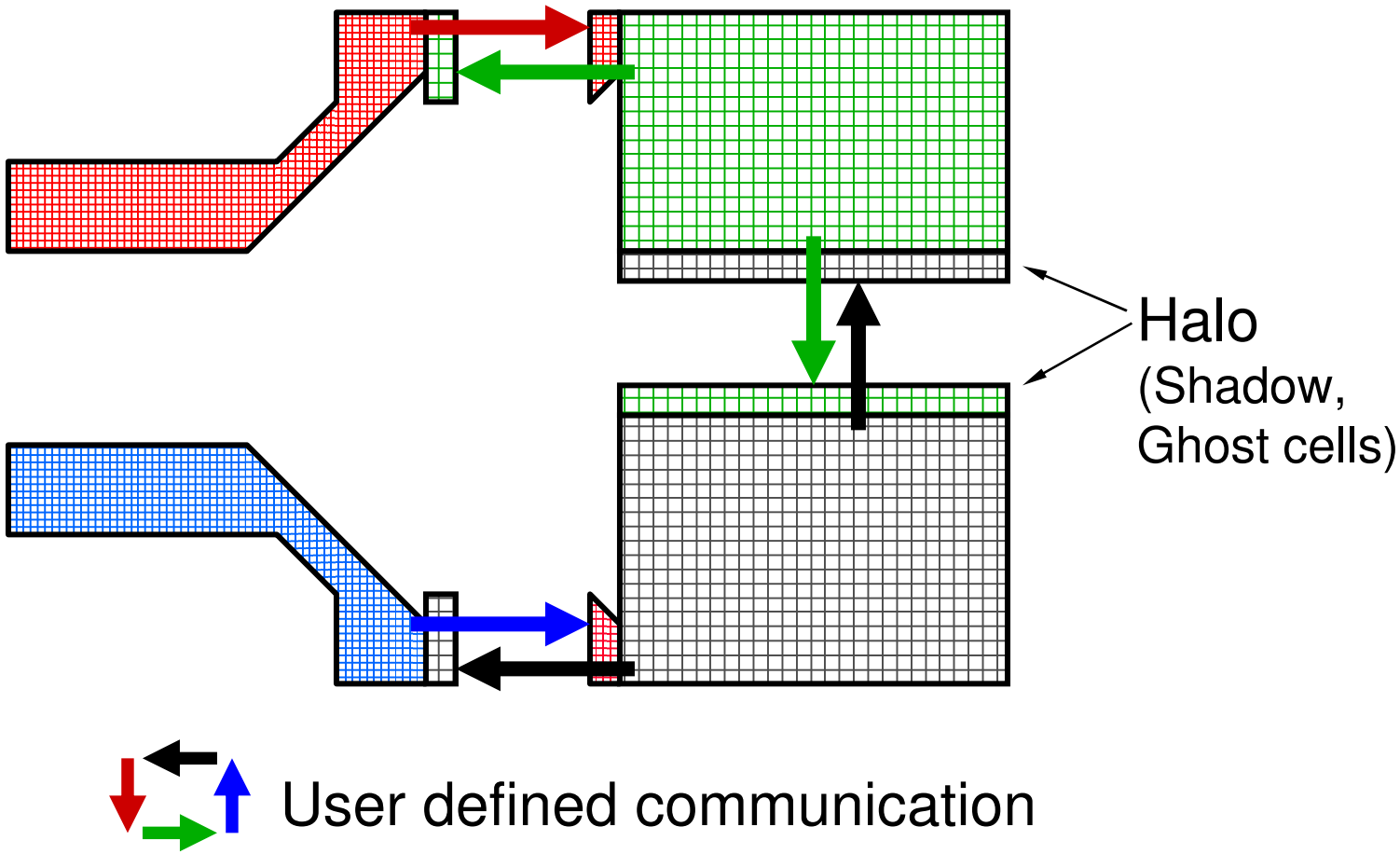
1. OpenMP
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. HPF (High Performance Fortran)
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (and synchronization) is implicit
3. MPI (Message Passing Interface)
   - User specifies how work & data is distributed
   - User specifies how and when communication has to be done
   - by calling MPI communication **library-routines**

Message Passing Program Paradigm – MPI, I.

- **Each processor in a message passing program runs a** *sub-program*
  - written in a conventional sequential language, e.g., C or Fortran,
  - typically the same on each processor (SPMD)
- **All work and data distribution is based on value of** *myrank*
  - returned by special library routine
- **Communication via special send & receive routines (***message passing***)**



Additional Halo Cells – MPI, II.



Message Passing – MPI, III.

```
Call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
Call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
m1 = (m+size-1)/size;   ja=1+m1*myrank;   je=max(m1*(myrank+1), m)
jax=ja-1;  jex=je+1   // extended boundary with halo

Real :: A(n, jax:jex), B(n, jax:jex)
do j = max(2,ja), min(m-1,je)
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```

- Data definition
- Loop over y-dimension
- Vectorizable loop over x-dimension
- Calculate B,
  - using upper and lower,
    - left and right value of A

```
Call MPI_Send(.......)   ! - sending the boundary data to the neighbors
Call MPI_Recv(.......)   ! - receiving from the neighbors,
                         !   storing into the halo cells
```

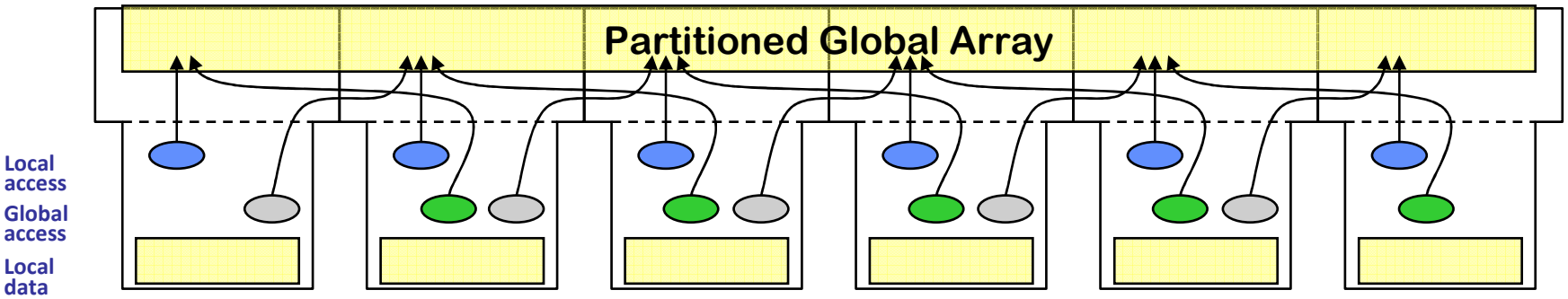
skipped

Summary — MPI, IV.

- **MPI (Message Passing Interface)**
  - standardized distributed memory parallelism with message passing
  - process-based
  - the user has to specify the <u>work distribution</u> & <u>data distribution</u> & all <u>communication</u>
  - synchronization implicit by completion of communication
  - the application processes are calling MPI library-routines
  - compiler generates normal sequential code
  - typically domain decomposition is used
  - communication across domain boundaries
  - standardized
    MPI-1: Version 1.0 (1994), 1.1 (1995), 1.2 (1997)
    MPI-2: since 1997
    MPI-2.1: June 2008

Programming styles with PGAS

- **Data is partitioned among the processes, i.e.,** without halos
  - Fine-grained access to the neighbor elements when needed
  - ➢ Compiler has to implement automatically (and together)
    - pre-fetches
    - bulk data transfer (instead of single-word remote accesses)
  - ➢ May be very slow if compiler fails
- **Application implements** halo **storage**
  - Application organizes halo updates with bulk data transfer
  - ➢ Advantage: High speed remote accesses
  - ➢ Drawbacks: Additional memory accesses and storage needs



Coming from MPI – what's different with PGAS?

```
size     = num_images()
myrank   = this_image() – 1
m1 = (m+size-1)/size;   ja=1; je= m1;   ! Same values on all processes
jax=ja-1;  jex=je+1   // extended boundary with halo
ja_loop=1; if(myrank==0) jaloop=2; jeloop=min((myrank+1)*m1,m–1) – myrank*m1;
Real :: A(n, jax:jex), B(n, jax:jex)
do j =  jaloop,  jeloop   ! Orig.: 2, m-1
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```

Data definition

Loop over y-dimension

Vectorizable loop over x-d

Calculate B,
using upper and lower,
left and right value of A

in original index range

remove range of lower processes

```
! Local halo  =  remotely computed data
  B(:,jex)  =  B(:,1)[myrank+1]
  B(:,jax)  =  B(:,m1)[myrank–1]
```

```
! Trick in this program:
! Remote memory access instead of
! MPI send and receive library calls
```

Let's start

The Process Model

- **Several processes are collectively started with a start-up tool**
  - Like mpirun or mpiexec
  - E.g., on Cray:
    aprun –N <processes per node> –n <total number of processes> ./<executable> <arg1> …
- **Processes are numbered:**
  - CAF: special intrinsic functions
    - numprocs = **num_images()**
    - myrank = **this_image() – 1** ! this_image() returns 1..num_images()
  - UPC: special global built-in "variables"
    - **#include <upc.h>**
    - **#include <upc_collective.h>**
    - numprocs = **THREADS**;
    - myrank = **MYTHREAD**; /* MYTHREAD yields 0..THREADS-1 */

To get myrank starting with 0

Default: Variables are local

- **Each process**
  - in CAF so-called "image"
  - in UPC so-called "thread"

  has its own data

skipped

Compilation and Execution

- **On Cray XT5m, hwwxt5.hww.de (at HLRS), with Cray compilers**
  - Initialization: `module switch PrgEnv-pgi PrgEnv-cray`
  - Compile:
    - UPC: `cc -h upc -h vector0 -l upc_open -o myprog myprog.c`
    - CAF: `crayftn -L/opt/xt-pe/2.1.50HD/lib_TV/snos64 -O vector0 -h caf -o myprog myprog.f90`

      Workaround for an installation problem
  - Execute (interactive test on 4 nodes with each **8** cores):
    - `qsub -I -lmppwidth=32,mppnppn=8, mem=1GB`
    - `aprun -n 32 -N 8 ./myprog` (all **8** cores per node are used)
    - `aprun -n 16 -N 4 ./myprog` (only 4 cores per node are used)

Compilation and Execution

- **On Cray XT4, franklin.nersc.gov (at NERSC), with PGI compiler**
  - **UPC only**
  - Initialization: `module switch PrgEnv-pgi PrgEnv-upc-pgi`
  - Compile:
    - UPC: `upcc -O -pthreads=4 -o myprog myprog.c`
  - Execute (interactive test on 8 nodes with each 4 cores):
    - `qsub -I -q debug -lmppwidth=32,mppnppn=4,walltime=00:30:00 -V`
    - `upcrun -n 32 -cpus-per-node 4 ./myprog`
    - Please use "debug" only with batch jobs, not interactively!
  - For the tutorial, we have a special queue: `-q special`
    - `qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V`
    - `upcrun -n 4 -cpus-per-node 4 ./myprog`
    - Limit: 30 users x 1 node/user

Workaround for an installation problem

Compilation and Execution

- **On Cray XT4, franklin.nersc.gov (at NERSC), with Cray compilers**
  - Initialization: `module switch PrgEnv-pgi PrgEnv-cray`
  - Compile:
    - UPC: `cc -h upc -h vector0 -l upc_open -o myprog myprog.c`
    - CAF: `crayftn -O vector0 -h caf -o myprog myprog.f90`
  - Execute (interactive test on 8 nodes with each **4** cores):
    - `qsub -I -q debug -lmppwidth=32,mppnppn=4,walltime=00:30:00 -V`
    - `aprun -n 32 -N 4 ./myprog` (all 4 cores per node are used)
    - `aprun -n 16 -N 2 ./myprog` (only 2 cores per node are used)
    - Please use "debug" only with batch jobs, not interactively!
  - For the tutorial, we have a special queue: `-q special`
    - `qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V`
    - `aprun -n 4 -N 4 ./myprog`
    - Limit: 30 users x 1 node/user

Practical 1: Each process writes its own rank and the total number of processes

- Login on your parallel system with your personal account:
  - **ssh –X franklin.nersc.gov**
- Copy skeletons+solutions into your working directory
  - **cp /tmp/rabenseifner_PGAS.tar.gz . ; tar –xzvf rabenseifner_PGAS.tar.gz**
- Change working directory to
  - **cd UPC or cd CAF**
  - **cat README** → you find all commands **UPC-pgi UPC-cray CAF-cray**
- **Initialize** your UPC or CAF environment, see README: **module switch …..**
- **Write a new UPC/CAF program that prints one line by each process:**
  - **cp hello_serial.c hello_pgas.c** or **cp hello_serial.f90 hello_pgas.f90**
- **With an if-clause, only process 0 should write an additional line:**
  - **"Hello world"**
- Compile and run according to the README
  - **qsub –vPROG=hello_pgas –q special qsub_1x4.sh**
- Desired output: →

```
I am 2 of 4
Hello world
I am 0 of 4
I am 3 of 4
I am 1 of 4
```

hello

https://fs.hlrs.de/projects/rabenseifner/publ/SciDAC2009-Part3-PGAS.pdf

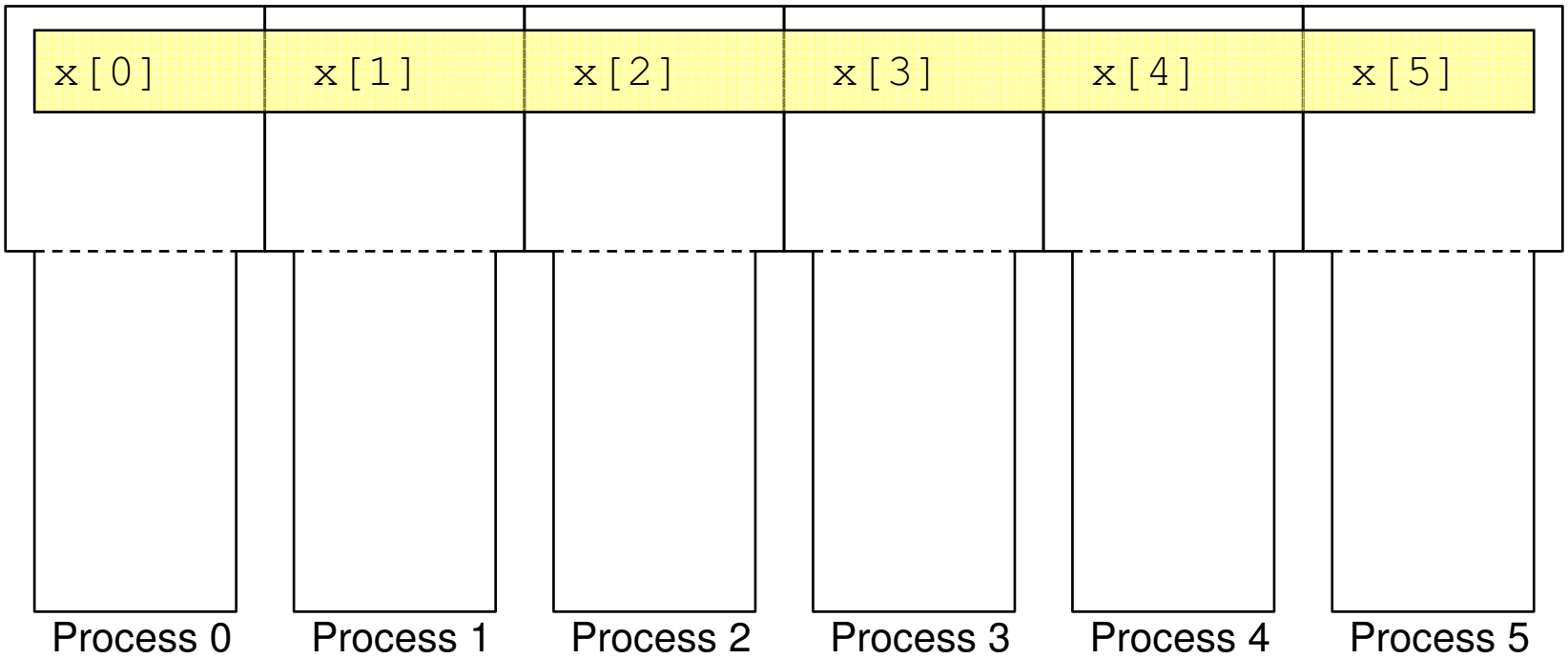
PGAS Data and Barrier Synchronization

Partitioned Global Address Space: Distributed variable

- **Declaration:**
  - UPC: `shared float x[THREADS];` //**statically allocated outside of functions**
  - CAF: `real :: x[0:*]`
- **Data distribution:**

UPC: "Parallel dimension"

CAF: "Co-dimension"



Local access to local part of distributed variables

- **UPC:**

```
shared float x[THREADS];
float *x_local;

x_local = (float *) &x[MYTHREAD];

*x now equals x[MYTHREAD]
```

- **CAF:** (0-based ranks)

```
real :: x[0:*]
numprocs=num_images()
myrank  =this_image()-1

x now equals x[myrank]
```

(1-based ranks)

```
real :: x[*]
numprocs=num_images()
myrank  =this_image()

x now equals x[myrank]
```

CAF-only: Multidimensional process numbering

- Co-dimensions may be multi-dimensional
- Each variable may use a different process numbering

```
integer :: numprocs, myrank, coord1, coord2, coords(2)
real :: x[0:*]
real :: y[0:1,0:*] !high value of last coord must be *

numprocs = num_images()
myrank   = this_image(x,1) ! x is 0-based
coord1   = this_image(y,1)
coord2   = this_image(y,2)
coords   = this_image(y)   ! coords-array!

X now equals x[myrank]
y now equals y[coord1,coord2]
         and y[coords(1),coords(2)]
```

Typical collective execution with access epochs

Process 0	Process 1	
UPC: *x_local = 17.0; CAF: x = 17.0	UPC: *x_local = 33.0; CAF: x = 33.0	Local accesses on PGAS data
Barrier synchronization	Barrier synchronization	Barrier until all processes have finished their local accesses
UPC: printf(… , x[1]) CAF: print *, x[1]	UPC: printf(… , x[0]) CAF: print *, x[0]	Remote accesses
Barrier synchronization	Barrier synchronization	Barrier until all processes have finished their remote accesses
UPC: x[0] = 29.0; CAF: x[0] = 29.0 …	UPC: x[1] = 78.0; CAF: x[1] = 78.0 …	Local accesses

Both notations are equivalent

Collective execution – same with remote write / local read

Process 0	Process 1	
UPC: x[1] = 33.0; CAF: x[1] = 33.0	UPC: x[0] = 17.0; CAF: x[0] = 17.0	Remote accesses on PGAS data
Barrier synchronization	Barrier synchronization	Barrier until all processes have finished their remote accesses
UPC: printf(…, *x_local) CAF: print *, x	UPC: printf(…, *x_local) CAF: print *, x	Local accesses
Barrier synchronization	Barrier synchronization	Barrier until all processes have finished their local accesses
UPC: x[1] = 78.0; CAF: x[1] = 78.0 …	UPC: x[0] = 29.0; CAF: x[0] = 29.0 …	Remote accesses

Synchronization

- Between a **write access** and a (subsequent or preceding) **read or write access** of the **same data** from **different processes**, a synchronization of the processes must be done!
- Most simple synchronization: barrier between all processes
- UPC:

```
Accesses to distributed data by some/all processes
upc_barrier;
Accesses to distributed data by some/all processes
```

- CAF:

```
Accesses to distributed data by some/all processes
sync all
Accesses to distributed data by some/all processes
```

Examples

- UPC:

write
sync
read

```
shared float x[THREADS];
x[MYTHREAD] = 1000.0 + MYTHREAD;
upc_barrier;
printf("myrank=%d, x[neighbor=%d]=%f\n",
 myrank, (MYTHREAD+1)%THREADS,
       x[(MYTHREAD+1)%THREADS]);
```

- CAF:

write
sync
read

```
real :: x[0:*]
integer :: myrank, numprocs
numprocs=num_images();  myrank  =this_image()-1
x = 1000.0 + myrank
sync all
print *, 'myrank=', myrank,
         'x[neighbor=', mod(myrank+1,numprocs),
         ']=', x[mod(myrank+1,numprocs)]
```

Practical 2: Each process writes neighbor data

- **Make a copy of your hello.c or hello.f90 program**
- **Declare a distributed variable x with integer type**
- **x = 1000+myrank**
- **Each process should print**
  myrank, numprocs, x[myrank–1], x[myrank], x[myrank+1]
- **With – 1 and +1 modulo numprocs:**
  - Caution: (myrank–1+numprocs)%numprocs due to handling of negatives
- **Output (red=modulo had to be applied):**

```
Number of processes = 4
myrank=2  x[1]=1001  x[2]=1002  x[3]=1003
myrank=0  x[3]=1003  x[0]=1000  x[1]=1001
myrank=3  x[2]=1002  x[3]=1003  x[0]=1000
myrank=1  x[0]=1000  x[1]=1001  x[2]=1002
```

neighbor_upc

neighbor_caf

Practical 2 – advanced exercise (a): Use of local short-cuts

- **Use local access methods instead of x[myrank]**
- **Same output**

```
Number of processes = 4
myrank=2  x[1]=1001  x[2]=1002  x[3]=1003
myrank=0  x[3]=1003  x[0]=1000  x[1]=1001
myrank=3  x[2]=1002  x[3]=1003  x[0]=1000
myrank=1  x[0]=1000  x[1]=1001  x[2]=1002
```

Practical 2 – advanced exercise (b-CAF): Two-dimensional process arrangement in CAF

- **Use a two-dimensional process layout with**
  - Zero-based coordinates
  - Size of first dimension = 2
  - Start always with an even number of processes
- **Print neighbor values in the diagonals (-1,-1) and (+1,+1)**
- **Output with CAF (sorted by myrank):**

myrank (if 0-based)
Example with 6 processes

c1 \ c2	0	1	2
0	**0**	**2**	**4**
1	**1**	**3**	**5**

```
Number of processes = 6 (with CAF)
myrank=0 c1=0 c2=0 x[1,2]=1005 x[0,0]=1000 x[1,1]=1003
myrank=1 c1=1 c2=0 x[0,2]=1004 x[1,0]=1001 x[0,1]=1002
myrank=2 c1=0 c2=1 x[1,0]=1001 x[0,1]=1002 x[1,2]=1005
myrank=3 c1=1 c2=1 x[0,0]=1000 x[1,1]=1003 x[0,2]=1004
myrank=4 c1=0 c2=2 x[1,1]=1003 x[0,2]=1004 x[1,0]=1001
myrank=5 c1=1 c2=2 x[0,1]=1002 x[1,2]=1005 x[0,0]=1000
```

Practical 2 – advanced exercise (b-UPC): Two-dimensional process arrangement in UPC

- **Use a two-dimensional process layout with**
  - Zero-based coordinates
  - Size of first dimension = 2
  - Start always with an even number of processes
- **Print neighbor values in the diagonals (-1,-1) and (+1,+1)**
- **Output with UPC (sorted by myrank):**

myrank (if 0-based)
Example with 6 processes

c1 \ c2	0	1	2
0	**0**	**1**	**2**
1	**3**	**4**	**5**

```
Number of processes = 6 (with UPC)
myrank=0 c1=0 c2=0 x[1,2]=1005 x[0,0]=1000 x[1,1]=1004
myrank=1 c1=0 c2=1 x[1,0]=1003 x[0,1]=1001 x[1,2]=1005
myrank=2 c1=0 c2=2 x[1,1]=1004 x[0,2]=1002 x[1,0]=1003
myrank=3 c1=1 c2=0 x[0,2]=1002 x[1,0]=1003 x[0,1]=1001
myrank=4 c1=1 c2=1 x[0,0]=1000 x[1,1]=1004 x[0,2]=1002
myrank=5 c1=1 c2=2 x[0,1]=1001 x[1,2]=1005 x[0,0]=1000
```

Work-Distribution and Other Synchronization Concepts

Work-distribution

- All work must be distributed among the processes (i.e., UPC threads or CAF images)
- All data must be distributed
- Parts of the data must be stored as PGAS data
- Global sum / minimum / maximum operations must be parallelized through **global reduction operations** (next slides)
- Work distribution must fit to data distribution

Manual Work-distribution

– Contiguous chunks of work (CAF example)

```
do i=1,niter
   …
end do
```

→

```
integer :: numprocs, myrank, chunk
numprocs       =  num_images()
myrank  =  this_image() - 1
chunk = (niter-1)/numprocs + 1   ! Rounding up
do i = 1+myrank*chunk, min(niter,(myrank+1)*chunk)
   …
end do
```

– Round robin distribution (UPC example)

```
for(i=0; i<niter; i++)
{
   …
}
```

→

```
for(i=MYTHREAD; i<niter; i=i+THREADS)
{
   …
}
```

Automatic Work-distribution

- **UPC only**
  - upc_forall (start_expr; condition; iteration_expr; **affinity**) **statement**
  - When **affinity**: → **statement** is executed by thread with rank
    - Integer → affinity modulo THREADS
    - pointer-to-shared → upc_threadof(affinity), e.g. `upc_threadof(&a[i])`
    - `continue` or empty → by all threads!
  - Execution of **statement** in each iteration must be independent.

```
for(i=0; i<niter; i++)
{ … }
```

→

```
upc_forall(i=0; i<niter; i++; i)
{ …  /* with round robin distribution */ }
```

```
for(i=0; i<niter; i++)
{ … }
```

→

```
upc_forall(i=0; i<niter; i++; i*THREADS/niter)
{ …  /* contiguous chunks of iterations */ }
```

  - Nested upc_forall are allowed:
    - The outermost with affinity not equal `continue` will be parallelize

Global reduction, e.g., max

Sequential code

```
global_max = 0
do i=1,100
  x = …
  global_max = max(x, global_max)
end do
```

Parallel code

```
real :: local_max[*]
integer :: rank
local_max = 0
do i=1,100/numprocs
  x = …
  local_max = max(x, local_max)
end do

sync all
if (myrank == 0) then
 global_max = 0
 do rank=0,numprocs-1
  global_max =
     max(local_max[rank], global_max)
 end do
 ! Preparation of broadcast
 !   of global_max to all processes
 local_max = global_max
end if
sync all
! Broadcast through remote reading
!   by all processes
global_max = local_max[0]
```

Original code + work distribution

- Each process stores local part into a PGAS array
- Synchronization
- Master process computes global result and
- stores it into local PGAS element
- Synchronization
- All other processes read this element
- ➢ global result is everywhere available

skipped

Same with min. code changes

Sequential code

```
global_max = 0
do i=1,100
  x = …
  global_max = max(x, global_max)
end do
```

Parallel code

```
real :: local_max[*]
integer :: rank
global_max = 0
do i=1,100/numprocs
  x = …
  global_max = max(x, global_max)
end do
local_max = global_max
sync all
if (myrank == 0) then
 global_max = 0
 do rank=0,numprocs-1
  global_max =
     max(local_max[rank], global_max)
 end do
 ! Preparation of broadcast
 !   of global_max to all processes
 local_max = global_max
end if
sync all
! Broadcast through remote reading
!   by all processes
global_max = local_max[0]
```

Original code + work distribution

- Each process stores local part into a PGAS array
- Synchronization
- Master process computes global result and
- stores it into local PGAS element
- Synchronization
- All other processes read this element
- ➢ global result is everywhere available

Critical Section

- **Principle:**

Begin of a critical section
    Code that is executed in all processes,
    only in one process at a time
End of this critical section

- **CAF:**

```
critical
        …  ! Code that is executed in all processes,
        …  ! only in one process at a time
end critical
```

CAF: Using a Critical Section

Sequential code

```
global_max = 0
do i=1,100
  x = …
  global_max = max(x, global_max)
end do
```

Parallel code

```
real :: global_max[*]
real :: local_max
local_max = 0
do i=1,100/numprocs
  x = …
  local_max = max(x, local_max)
end do

if (myrank == 1) then
  global_max=0   ! or smallest number
end if
sync all ! Don't proceed on others
         ! until process 1 has done
critical
  global_max[1] = max(local_max, &
                &      global_max[1])
end critical
sync all ! Don't proceed on proc.1
         ! until other proc. have done
global_max = global_max[1]
```

Original code
+ work distribution

Using Locks for Critical Sections

- **UPC:**

```
upc_lock_t *lock;

lock = upc_all_lock_alloc(); /* collective call
                                only once */

upc_barrier;
upc_lock(lock);
        … ! Code that is executed in all processes,
        … ! only in one process at a time
upc_unlock(lock);
```

UPC: Locks

Sequential code

```
#define max(a,b) ((a)>(b) ? a : b)
global_max = 0;
for (i=1; i<=100; i++)
  x = … ;
  global_max = max(x, global_max);
end do
```

Parallel code

```
static shared double global_max;
upc_lock_t *lock;
lock = upc_all_lock_alloc();

local_max = 0;
for (i=1; i<=100/numprocs; i++)
  x = …
  local_max = max(x, local_max)
end do

if (myrank == 0) global_max=0;
upc_barrier;
upc_lock(lock);
 global_max=max(local_max,global_max);
upc_unlock(lock);
upc_barrier;
```

Original code
+ work distribution

Collective intrinsic routines: Reduction with UPC


```
void upc_all_reduceT(shared void * restrict dst,
                     shared const void * restrict src,
                     upc_op_t op,  size_t nelems,
                     size_t blk_size,
                     TYPE(*func)(TYPE, TYPE),
                     upc_flag_t flags);
```

see next chapter, default = 1

UPC_IN_NOSYNC | UPC_OUT_NOSYNC

op	Operation
UPC_ADD	Addition.
UPC_MULT	Multiplication.
UPC_MIN	For all data types, find the minimum value.
UPC_MAX	For all data types, find the maximum value.
UPC_AND	Bitwise AND for integer and character variables.
UPC_OR	Bitwise OR for integer and character variables.
UPC_XOR	Bitwise XOR for integer and character variables.
UPC_LOGAND	Logical AND for all variable types.
UPC_LOGOR	Logical OR for all variable types.
UPC_FUNC	Use the specified commutative function func to operate on the data in the src array at each step.
UPC_NONCOMM_FUNC	Use the specified non-commutative function func to operate on the data in the src array at each step.

T	TYPE	T	TYPE
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

UPC: Reduction routine

Sequential code

```
#define max(a,b) ((a)>(b) ? a : b)
global_max = 0;
for (i=1; i<=100; i++)
  x = … ;
  global_max = max(x, global_max);
end do
```

- **(not yet available with CAF)**

Parallel code

```
#define BLK_SIZE 1
#define NELEMS 1  /* per thread */
shared [BLK_SIZE] double
    local_max[NELEMS*THREADS];

local_max = 0;
for (i=1; i<=100/numprocs; i++)
  x = …
  local_max[MYTHREAD]
    = max(x, local_max[MYTHREAD])
end do

upc_barrier;
upc_all_reduceD(&global_max,local_max,
   UPC_MAX, NELEMS, BLK_SIZE, NULL,
   UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

Original code
+ work distribution

skipped

CAF: Reduction routine (not part of Fortran 2008)

Sequential code

```
global_max = 0
do i=1,100
  x = …
  global_max = max(x, global_max)
end do
```

Parallel code

```
real :: global_max
real :: local_max[*]
local_max = 0
do i=1,100/numprocs
  x = …
  local_max = max(x, local_max)
end do

call co_sum(local_max, global_max)
```

Original code + work distribution

- **Such collective communication routines**
  - were part of ISO/IEC JTC1/SC22/WG5 N1708, Jan. 2008.
  - On the joint meeting of WG5 and PL22.3, Las Vegas, USA, February 10-15,
    they were moved into a separate Technical Report on "Enhanced Parallel Computing Facilities".

Practical 3: Pi

- Make a copy of ../skel/pi_serial.* into your working directory
- cp pi_serial.c pi_upc.c
  cp pi_serial.f90 pi_caf.f90
- Parallelize the loop (work-distribution)
- Implement the global sum (reduction operation)
- Result may differ in the last 4 digits due to different rounding errors.

pi_upc

pi_caf

Practical 3 – advanced exercise: Pi

- **Use several methods for the work-distribution.**
- **Use several methods for the global sum (reduction operation).**
- **Where are performance problems.**

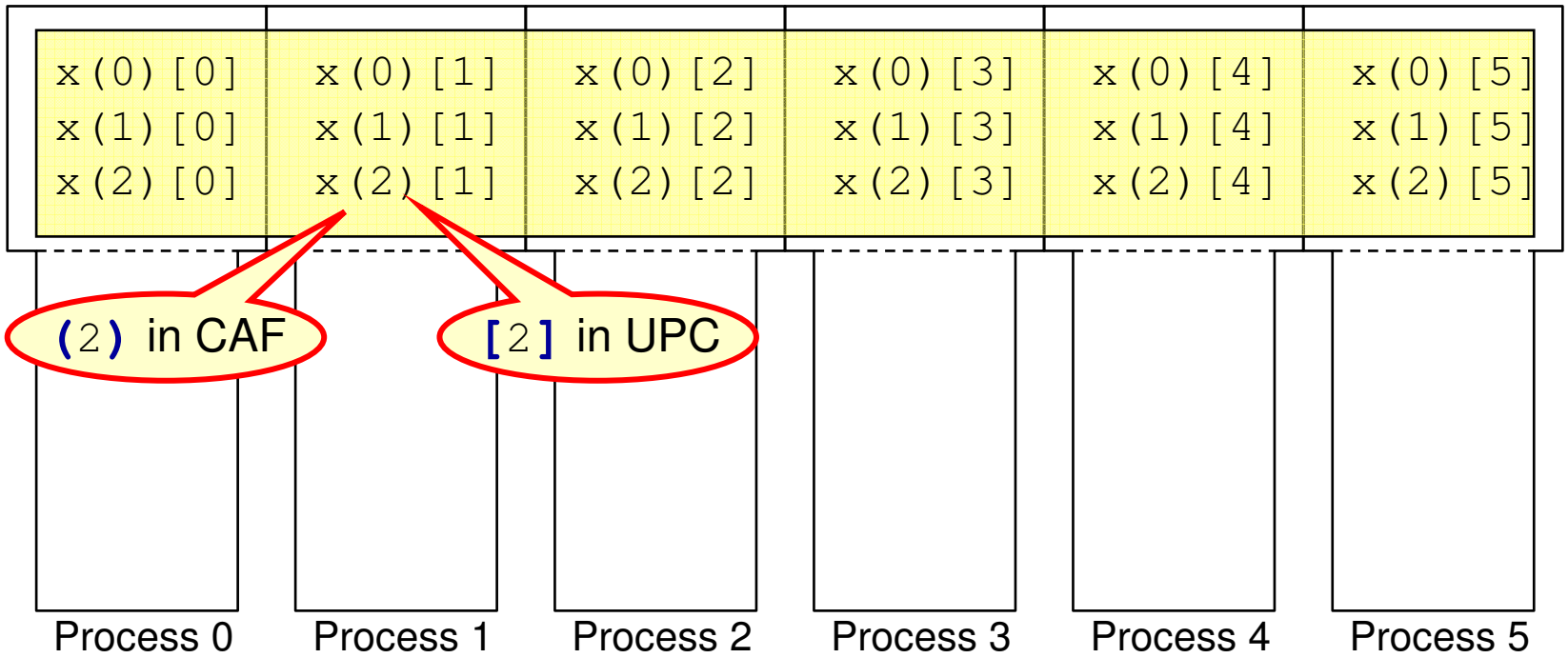
Arrays and Pointers

Partitioned Global Address Space: Distributed arrays

- **Declaration:**
  - UPC: `shared float x[3][THREADS];` //**statically allocated**
  - CAF: `real :: x(0:2)[0:*]`
- **Data distribution:**

UPC: "Parallel dimension"

CAF: "Co-dimension"



skipped

Restrictions with CAF

- **Automatic co-arrays (→ see slide "Dynamic allocation with CAF")**

```
subroutine sub(n)
  integer :: n
  real    :: work(n)[*] ! NOT supported: automatic co-array
```

- **Dummy arguments with assumed shape or co-shape**

```
subroutine sub(x)
  real    :: x(:)[*]    ! NOT supported: assumed shape co-array
```

- **Pointer co-arrays**

```
real,pointer :: ptr[*]  ! NOT supported: pointer co-array
```

- **Co-array as parameter**

```
real,parameter :: pi[*] ! NOT supported: parameter co-array
```

skipped

Restrictions with CAF (continued)

- **Co-arrays with different extent on different processes**

```
subroutine sub(n)
  integer :: n
  real,allocatable:: work(:)[*] ! Supported: allocatable co-arr
  allocate( a(1:5+myrank)[*] )  ! NOT supported: different size
```

- **Co-arrays as components of derived types**

```
type buftype
  real    :: x(5)[*]    ! NOT supported: co-array as components
end type buftype
type(buftype) :: buf
```

- **Sub-objects of co-arrays in equivalence and data statements**

```
real :: aco(10)[*], bco(10)[*], c(10)
equivalence (aco(5)   ,bco(1)   )  ! Permitted equivalence
equivalence (aco(5)[2],bco(1)[3])  ! NOT supported
equivalence (aco(5)   ,c(1)     )  ! NOT supported
data         aco(1)[3]  /7.5/      ! NOT supported
```

skipped

Supported with CAF

- **Co-arrays of derived types**

```
type buftype
  pointer :: ptr
  real    :: x(5), y(5)
end type buftype
type(buftype) :: buf(7)[*]
```

Dynamic allocation with CAF

- **Allocable arrays can have co-dimensions:**

```
real,allocatable :: a(:,:)[:] ! Example: Two-dim. + one co-dim.
allocate( a(0:m,o:n)[0:*] )   ! Same m,n on all processes
```

- **Same shape on all processes is needed!**

```
real,allocatable :: a(:)[:]          ! WRONG example
allocate( a(myrank:myrank+1)[0:*] ) ! NOT supported
```

Distributed arrays with UPC

- UPC shared objects must be statically allocated
- Definition of shared data:
  - **shared** [**blocksize**] type variable_name;
  - **shared** [**blocksize**] type array_name[dim1];
  - **shared** [**blocksize**] type array_name[dim1][dim2];
  - …
- Default: blocksize=1
- The distribution is always round robin with chunks of **blocksize** elements
- Blocked distribution is implied if last dimension = THREADS and blocksize==1

the dimensions define which elements exist

See next slides

UPC shared data – examples

```
shared [1] float a[20];   // or
shared     float a[20];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0]	a[1]	a[2]	a[3]
a[4]	a[5]	a[6]	a[7]
a[8]	a[9]	a[10]	a[11]
a[12]	a[13]	a[14]	a[15]
a[16]	a[17]	a[18]	a[19]

```
shared [1] float a[5][THREADS];
// or
shared     float a[5][THREADS];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]

```
shared [5] float a[20];   // or
define N 20
shared [N/THREADS] float a[N];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0]	a[5]	a[10]	a[15]
a[1]	a[6]	a[11]	a[16]
a[2]	a[7]	a[12]	a[17]
a[3]	a[8]	a[13]	a[18]
a[4]	a[9]	a[14]	a[19]

THREADS=1st dim!

identical at compile time

```
shared [5] float a[THREADS][5];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0][0]	a[1][0]	a[2][0]	a[3][0]
a[0][1]	a[1][1]	a[2][1]	a[3][1]
a[0][2]	a[1][2]	a[2][2]	a[3][2]
a[0][3]	a[1][3]	a[2][3]	a[3][3]
a[0][4]	a[1][4]	a[2][4]	a[3][4]

UPC shared data – examples (continued)

```
shared float a[THREADS]; // or
shared [1] float a[THREADS];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0]	a[1]	a[2]	a[3]

```
shared float a;
// located only in thread 0
```

Thread 0	Thread 1	Thread 2	Thread 3
a			

```
shared [2] float a[20];
```

upc_threadof(&a[15]) == 3

Thread 0	Thread 1	Thread 2	Thread 3
a[0] a[1] a[8] a[9] a[16] a[17]	a[2] a[3] a[10] a[11] a[18] a[19]	a[4] a[5] a[12] a[13]	a[6] a[7] a[14] a[15]

Blank blocksize → located only in thread 0

```
shared [ ] float a[10];
```

Thread 0	Thread 1	Thread 2	Thread 3
a[0] a[1] a[2] … a[9]			

Pointer with UPC

- **Pointers themselves can be private or shared**
- **Pointers can point to private or shared data**

`float *p1;`	**Private** pointer pointing to a float in **private** memory
`shared float *p2;`	**Private** pointer pointing to a float in **shared** memory
`float *shared p3;`	**Shared** pointer pointing to a float in **private** memory (should not be used – senseless for other threads)
`shared float *shared p4;`	**Shared** pointer pointing to a float in **shared** memory

Pointer to local portions of shared data

- **P1-type pointer is valid.**

```
shared float a[5][THREADS];
float *a_local;

a_local = (float *) &a[0][MYTHREAD];

a_local[0] is identical with a[0][MYTHREAD]
a_local[1] is identical with a[1][MYTHREAD]
…
a_local[4] is identical with a[4][MYTHREAD]
```

- **May have performance advantages**

UPC Dynamic Memory Allocation

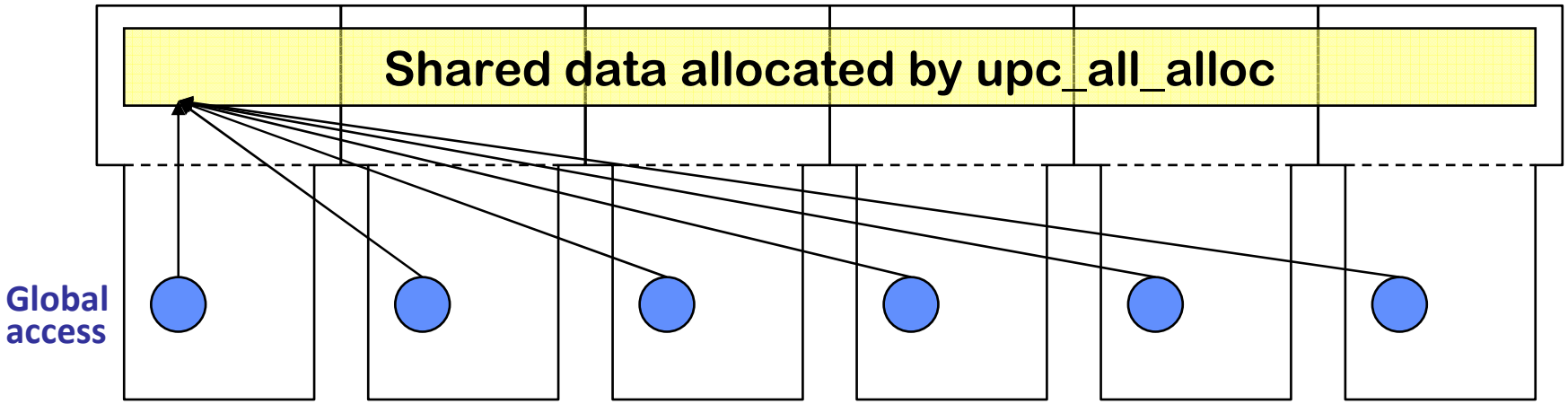
- **upc_all_alloc**
  - Collective over all threads (i.e., all threads must call)
  - All threads get a copy of the same pointer to shared memory

```
shared void *upc_all_alloc( size_t nblocks, size_t nbytes)
```

Run time arguments

  - Similar result as with static allocation at compile time:

```
shared [nbytes] char[nblocks*nbytes];
```

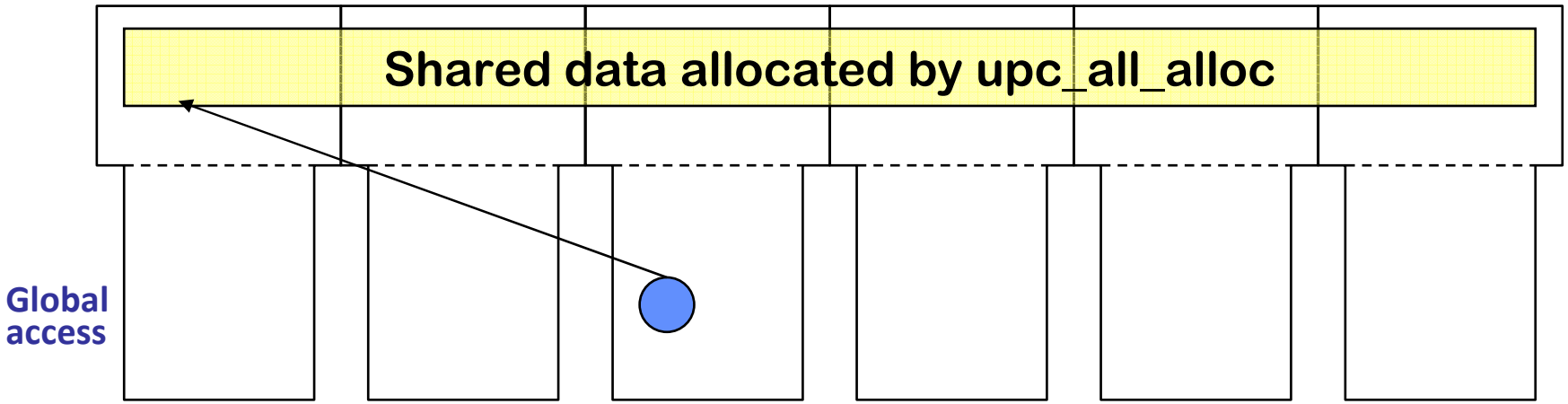


UPC Dynamic Memory Allocation (continued)

- **upc_global_alloc**
  - Only the calling thread gets a pointer to shared memory

```
shared void *upc_global_alloc( size_t nblocks, size_t nbytes)
```

  - Blocksize nbytes other than 1 is currently not supported
    - At least with Berkley UPC



skipped

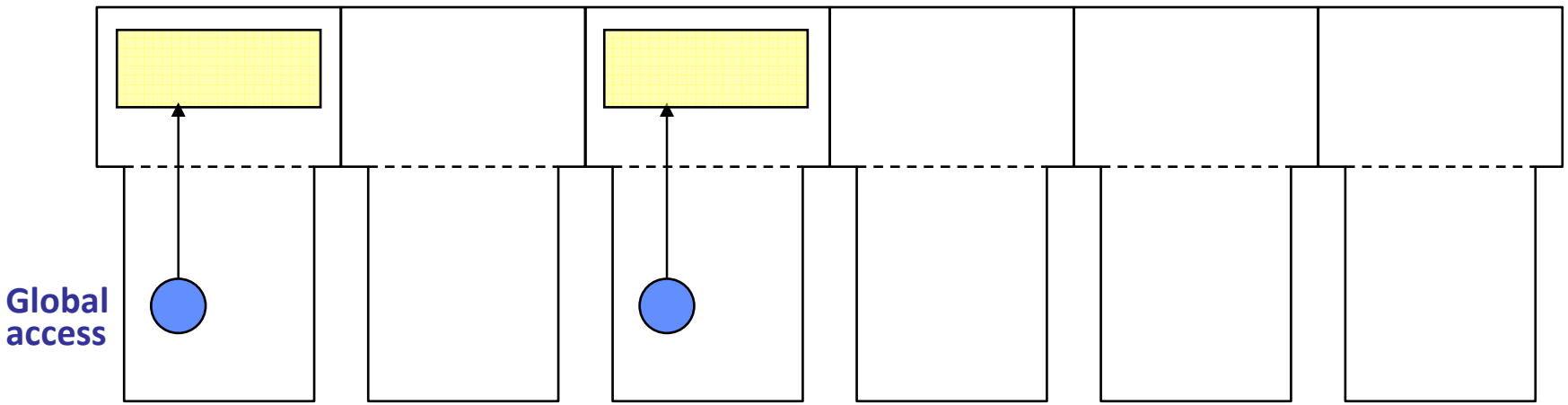
UPC Dynamic Memory Allocation (continued, 2nd)

- **upc_alloc**
  - Allocates memory in the local thread that is accessible by all threads
  - Only in calling processes

```
shared void *upc_alloc( size_t nbytes )
```

  - Similar result as with static allocation at compile time:

```
shared [] char[nbytes]; // but with affinity to the calling thread
```



skipped

UPC example with dynamic allocation


```
shared [] float * shared p4[THREADS];  // shared pointer array
                                       // to shared data
float *p1; // private pointer to private portion

int main(int argc, char **argv)
{ int i, n, rank;
  n = atoi(argv[1])
  p4[MYTHREAD] = (shared [] float *) upc_alloc(n * sizeof(float));
  p1 = (float *) p4[MYTHREAD];
  for (i=0; i<n; i++) {
    p1[i] = …
  }
  upc_barrier;
  if (MYTHREAD == 0) {
    for (rank=0; rank<THREADS; rank++)
      for (i=0; i<n; i++) {
        printf(……, p4[rank][i]);
      }
    }
  }
}
```

skipped

UPC example with type-p2 pointers


```
shared [] float * shared p4[THREADS]; // shared pointer array
                                      // to shared data
float *p1; // private pointer to private portion hared
shared [] float *p2_neighbor; // private pointer to shared data
int main(int argc, char **argv)
{ int i, n, rank, next;
  n = atoi(argv[1])
  p4[MYTHREAD] = (shared [] float *) upc_alloc(n * sizeof(float));
  p1 = (float *) p4[MYTHREAD];
  upc_barrier;

  next = MYTHREAD+1 % THREADS;
  p2_neighbor = p4[next];
  for (i=0; i<n; i++) {
    x[i] = …
  }
  upc_barrier;

  for (i=0; i<n; i++) {
    printf(……, p2_neighbor[i]);
  }
}
```

skipped

Practical 4: Heat

- **Make a copy of ../skel/heat_serial.* into your working directory**
- **cp heat_serial.c heat_upc.c**
  cp heat_serial.F90 heat_caf.F90
- **Parallelization is done in several steps:**
  - Domain (data) decomposition
  - With halo
  - Work decomposition
  - Abort criterion with reduction operation
  - Halo exchange
  - Print routine with global access

skipped

Practical 4: Heat – advanced (UPC only)

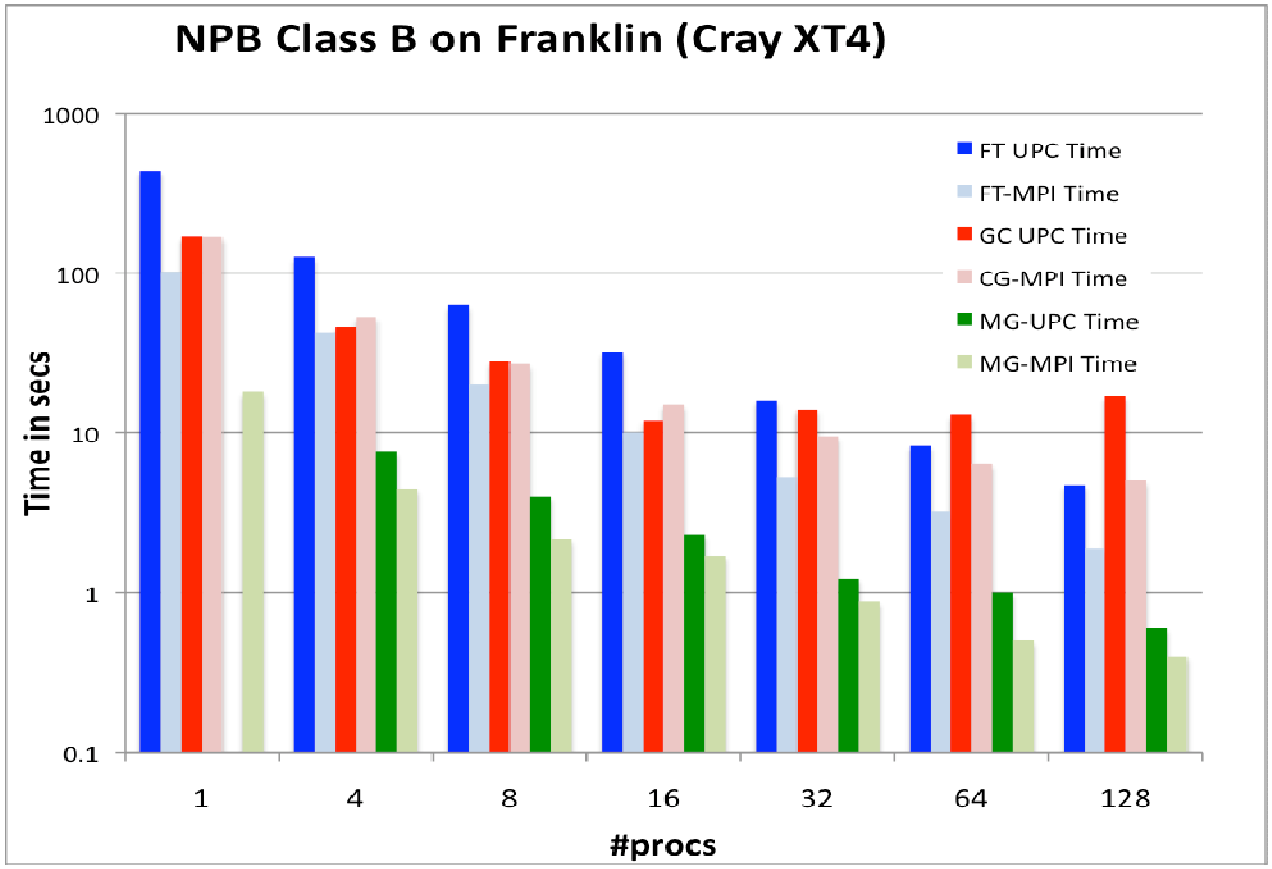
- **Usage of memory copy routines:**

```
upc_memcopy(destination, source, size);
  // copies shared data to shared data

upc_memput( destination, source, size);
  // copies private date to shared data

upc_memget( destination, source, size);
  // copies shared date to private data

upc_memset( destination, chr, size);

  // initializes shared data with a character (byte)
```

- **Substitute the exchange of contiguous halos with upc_memget**
- **Substitute the exchange of strided halos by**
  - Copying strided original data into contiguous scratch arrays
  - upc_memget into second scratch array
  - Copying back into strided halo array

Wrap up & Summary

Comparison of MPI and UPC – NAS Parallel Benchmarks



- Benchmarks show similar behavior
- FT, MG scalability ok in both implementations
  - Note: FT compiled with –g ! Coredump otherwise.
- CG poor scalability: irregular memory access, global sums for dot-products

skipped

NPB UPC vs MPI on Cray XT4

NPB-UPC 110404

UPC Compile and build:

```
upcc -O -T${NP}
```

```
This is upcc (the Berkeley Unified
Parallel C compiler), v. 2.8.0
```

Run:

```
setenv UPC_SHARED_HEAP_SIZE 2GB
setenv GASNET_MAX_SEGSIZE 4GB
aprun -n ${NP} -N 1 ft.B.${NP}
```

NPB3.2 MPI

MPI Compile and build:

```
ftn -O
```

```
pgf90 8.0-4 64-bit target on
x86-64 Linux -tp k8-64e
```

Run:

```
aprun -n ${NP} -N 1 ft.B.${NP}
```

Compilation for tracing with upc_trace:

```
upcc -g
```

Execution for tracing:

```
upcrun -trace -N 8 -c 1 -n 8 ./ft.B.8
```

Report generation:

```
upc_trace -t upc_trace.ft.B.8*
```

skipped

NPB UPC vs MPI Performance Statistics

NPB-UPC 110404

ft.B.8

```
total   =     63.422
fft     =     60.125
evolve  =      2.337
fftlow  =     39.842
fftcopy  =    10.441
transpose  =   9.805
all_to_all  = 2.658
```

Note: Compiled without optimization!
upcc –g -T${NP}

NPB3.2 MPI

ft.B.8

```
Total     : 20.683184
fft       : 19.251469
evolve    :  1.268715
fftlow    :  7.264877
fftcopy   :  2.830075
transpose:   9.132087
transpose1_loc: 6.315235
transpose1_glo:  1.793118
transpose1_fin:  1.017136
```

Note: Compiled with optimization!
ftn -O

fft: Computation a lot faster for MPI: Higher compiler optimization was possible for the MPI code.
Transpose: Communication time the same for both implementations.

skipped

NPB UPC upc_trace statistics

Source code snippet:

```
upc_barrier;
/* XXX Fortran version uses an MPI_alltoall() here */
for (i = 0; i < THREADS; i++)
  {
    upc_memget( (dcomplex *)&dst[MYTHREAD].cell[chunk*i],
                &src[i].cell[chunk*MYTHREAD],
                sizeof( dcomplex ) * chunk );
  }
upc_barrier;
```

Example Report

```
PUT REPORT:
SOURCE          LINE    TYPE        MSG:(min    max     avg     total)     CALLS
==============================================================================
ft.c             1952     GLOBAL  8.00 M    8.00 M    8.00 M    9.62 G   1232
 Thread 0                         8.00 M    8.00 M    8.00 M    1.20 G    154
 Thread 1                         8.00 M    8.00 M    8.00 M    1.20 G    154
 Thread 2                         8.00 M    8.00 M    8.00 M    1.20 G    154
 Thread 3                         8.00 M    8.00 M    8.00 M    1.20 G    154

BARRIER REPORT:
SOURCE          LINE    TYPE        MSG:(min    max     avg     total)     CALLS
==============================================================================
ft.c             1955       WAIT    20.1 us   61.4 ms   21.2 ms    3.7  s
    Thread 0..0                     20.1 us   18.1 ms    4.8 ms  105.3 ms     22
    Thread 1..1                      4.7 ms   21.1 ms   12.3 ms  270.8 ms     22
    Thread 2..2                     22.8 ms   39.8 ms   30.9 ms  680.3 ms     22
    Thread 3..3                     20.3 us   20.7 ms   12.2 ms  269.1 ms     22
```

Summary

- **Compiler exist**
- **Easier to program than MPI**
- **Latency hiding and bandwidth optimization of the compiler still weak**

MPI:

On a given system: **Optimization** of the …	
Numerics	Communication
Compiler A	MPI lib 1
Compiler B	MPI lib 2
Compiler C	MPI lib 3
---	MPI lib 4

Optimum is one of 3x4 choices

PGAS:

On a given system: **Optimization** of the …	
Numerics	Communication
Compiler A	with comm. A
Compiler B	with comm. A
Compiler C	with comm. A
---	---

Optimum is one of 3 choices

- **Significant opportunities for latency hiding, memory bandwidth optimization, …**

Annex

README – UPC on Cray XT…: PrgEnv-upc-pgi


```
Initialization:  module switch PrgEnv-pgi PrgEnv-upc-pgi

Interactive PBS shell:
In general:
  qsub -I -q debug   -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
In the SciDAC tutorial
  qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V

Again to the working directory:
  cd $PBS_O_WORKDIR

Compilation:
  upcc -O -pthreads=4 -o myprog myprog.c

Parallel Execution:
  upcrun -n 1 -cpus-per-node 4 ./myprog
  upcrun -n 2 -cpus-per-node 4 ./myprog
  upcrun -n 4 -cpus-per-node 4 ./myprog

If your batch job was initiated
with 8 cores, i.e., with: -lmppwidth=8,mppnppn=4
  upcrun -n 8 -cpus-per-node 4 ./myprog
```

>Pract.1

>Pract.2

>Pract.3

>Pract.4

README – UPC on Cray XT…: PrgEnv-upc-cray


```
Initialization:  module switch PrgEnv-pgi PrgEnv-cray

Interactive PBS shell:
In general:
  qsub -I -q debug   -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
In the SciDAC tutorial
  qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V

Again to the working directory:
  cd $PBS_O_WORKDIR

Compilation:
  cc -h upc -h vector0 -o myprog myprog.c

Parallel Execution:
  aprun -n 1 -N 1 ./myprog
  aprun -n 2 -N 2 ./myprog
  aprun -n 4 -N 4 ./myprog

If your batch job was initiated
with 8 cores, i.e., with: -lmppwidth=8,mppnppn=4
  aprun -n 8 -N 4 ./myprog
```

Compiler version 7.0 …?

Workaround for a correctness problem

>Pract.1

>Pract.2

>Pract.3

>Pract.4

README – UPC on Cray XT…: PrgEnv-caf-cray


```
Initialization:  module switch PrgEnv-pgi PrgEnv-cray

Interactive PBS shell:
In general:
  qsub -I -q debug   -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
In the SciDAC tutorial
  qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V

Again to the working directory:
  cd $PBS_O_WORKDIR

Compilation:
  crayftn -L/opt/xt-pe/2.1.50HD/lib_TV/snos64  -O vector0
          -hcaf -o myprog myprog.f90

Parallel Execution:
  aprun -n 1 -N 1 ./myprog
  aprun -n 2 -N 2 ./myprog
  aprun -n 4 -N 4 ./myprog

If your batch job was initiated
with 8 cores, i.e., with: -lmppwidth=8,mppnppn=4
  aprun -n 8 -N 4 ./myprog
```

Compiler version 7.0 …?

Workaround for a library path problem

Workaround for a correctness problem

>Pract.1

>Pract.2

>Pract.3

>Pract.4

hello_upc.c and hello_caf.f90


```
#include <upc.h>
#include <stdio.h>
int main(int argc, char** argv)
{
  if (MYTHREAD == 0) printf("hello world\n");
  printf("I am thread number %d of %d threads\n",
                        MYTHREAD,   THREADS);
  return 0;
}
```

```
program hello
implicit none
integer :: myrank, numprocs
myrank   = THIS_IMAGE() - 1
numprocs = NUM_IMAGES()
if (myrank == 0) print *, 'hello world'
write (*,*) 'I am image number',myrank, &
          & ' of ',numprocs,' images'
end program hello
```

>Pract.1

neighbor_upc.c

```
#include <upc.h>
#include <stdio.h>
shared int x[THREADS];
int main(int argc, char** argv)
{
  int myrank, left, right, numprocs;
  numprocs = THREADS;
  myrank = MYTHREAD;
  if (MYTHREAD == 0)
        printf("Number of processes = %d\n", numprocs);
  left  = (myrank-1+numprocs) % numprocs;
  right = (myrank+1) % numprocs;
  x[myrank] = 1000 + myrank;
  upc_barrier;
  printf("myrank=%d  x[%d]=%d  x[%d]=%d  x[%d]=%d\n",
       myrank,
       left, x[left], myrank, x[myrank], right, x[right]);
  return 0;
}
```

PGAS data

Each process stores its own value

Waiting until all processes have stored their x value

>Pract.2

neighbor_caf.f90

```
program neighbor
implicit none
integer :: x[0:*]
integer :: myrank, left, right, numprocs
myrank   = THIS_IMAGE() - 1
numprocs = NUM_IMAGES()
if (myrank == 0) print *, 'Number of processes =`,numprocs
left  = mod(myrank-1+numprocs, numprocs)
right = mod(myrank+1, numprocs)
x = 1000 + myrank;

SYNC ALL
write (*,*) 'myrank=',myrank, &
          & ' x[',left,  ']=', x[left],   &
          & ' x[',myrank,']=', x[myrank], &
          & ' x[',right, ']=', x[right]
end program neighbor
```

PGAS data

Each process stores its own value.
x is an optimized shortcut for x[myrank]

Waiting until all processes
have stored their x value

>Pract.2

pi_upc.c
Part 1 – the numerical loop


```
#include <upc.h>
…
shared double mypi[THREADS];
int main(int argc, char** argv)
{
…
/* calculate pi = integral [0..1] 4/(1+x**2) dx */
  w=1.0/n;
  sum=0.0;
  ichunk=(n-1)/THREADS+1;   /* = n/THREADS rounded up */
  ibegin=1+ichunk*(MYTHREAD);
  iend  =ibegin+ichunk-1;
  if(iend>n) iend=n;
  /*originally for(i=ibegin; i<=iend; i++) */
  for (i=ibegin; i<=iend; i++)
  /* or alternatively: for (i=1+MYTHREAD; i<=n; i=i+THREADS) */
  {
    x=w*((double)i-0.5);
    sum=sum+4.0/(1.0+x*x);
  }
  mypi[MYTHREAD]=w*sum;   /* originally: pi=w*sum; */
```

Local pi computation, PGAS array is needed for global summing up

Work-distribution

Each process can calculate only a partial sum

→ next slide

pi_upc.c
Part 2 – summing up


```
  upc_barrier;

  if (MYTHREAD==0)
  {
    pi=0.0;
        for (i=0; i<THREADS; i++)
        {
          pi = pi+mypi[i];
    }
  }
…
  if (MYTHREAD==0)
  {
    printf( "computed  pi      = %19.16f\n", pi );
…
  }
  return 0;
}
```

Waiting until all processes have stored their mypi result

Process 0 is calculating the total sum.
For larger process numbers,
a binary tree algorithm is needed.

Access to other processes' data

I/O only by process 0

>Pract.3

pi_caf.f90
Part 1 – the numerical loop


```
…
! additional variables for the manual CAF worksharing:
integer              :: myrank, numimages, ibegin, iend, ichunk
! distributed array for the partial result of each image:
real(kind=8)         :: mypi[*]
myrank     = THIS_IMAGE()
numimages  = NUM_IMAGES()
…
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n
sum=0.0_8
ichunk=(n-1)/numimages+1      ! = n/numimages rounded up
ibegin=1+ichunk*(myrank-1)
iend  =min(ichunk*myrank, n)
do i=ibegin,iend
! or alternatively:
! do i=1+(myrank-1), n, numimages
     x=w*(i-0.5_8)
!    function to integrate
     sum=sum + 4.0_8/(1.0_8+x*x)
enddo
mypi=w*sum
```

→ next slide

Local pi computation,
PGAS array is needed for global summing up

Work-distribution

Each process can calculate only a partial sum.
mypi is an optimized shortcut for mypi[myrank]

pi_caf.f90
Part 2 – summing up


```
SYNC ALL

if (myrank == 1) then
  pi=0.0
  do i=1,numimages
    pi = pi+mypi[i]
  end do
end if
…
if (myrank == 1) then
  write (*,'(a,f24.16)')   'computed  pi      = ', pi
  …
end if

end program compute_pi
```

Waiting until all processes have stored their mypi result

Process 0 is calculating the total sum.
For larger process numbers,
a binary tree algorithm is needed.

Access to other processes' data

I/O only by process 0

>Pract.3

Annex: Abstract

Multicore systems have begun to dominate the available compute platforms for HPC applications. These new systems are causing application designers to consider alternate programming models to pure MPI, including **Hybrid OpenMP/MPI models** and **PGAS (Partitioned Global Address Space) languages**. In this tutorial we give an introduction to these alternate models and consider their performance on recent multicore machines. We overview the common current architectures and give specifics on how to run these languages on machines commonly available to SciDAC projects. We use benchmarks to compare various machines as well as the languages. We also briefly describe some of the possible upcoming platforms that may become available to including cloud computing offerings.