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

Alice Koniges – NERSC, Lawrence Berkeley National Laboratory (LBNL)
Katherine Yelick – University of California, Berkeley and LBNL
Rolf Rabenseifner – High Performance Computing Center Stuttgart (HLRS), Germany
Reinhold Bader – Leibniz Supercomputing Centre (LRZ), Munich/Garching, Germany
David Eder – Lawrence Livermore National Laboratory
Filip Blagojevic, Robert Preissl and Paul Hargrove – Lawrence Berkeley National Laboratory

A full-day tutorial at SC12,
November 12, 2012, Salt Lake City, Utah , USA

Outline

- **Basic PGAS concepts** (Katherine Yelick)
  - Execution model, memory model, resource mapping, …
  - Standardization efforts, comparison with other paradigms

  → Exercise 1 (hello)
- **UPC and CAF basic syntax** (Rolf Rabenseifner)
  - Declaration of shared data / coarrays, synchronization
  - Dynamic entities, pointers, allocation

  → Exercise 2 (triangular matrix)
- **Advanced synchronization concepts** (Reinhold Bader)
  - Locks and split-phase barriers, atomic procedures, collective operations
  - Parallel patterns

  → Exercises 3+4 (reduction+heat)
- **Applications, Optimization, and Hybrid Programming** (Alice Koniges, David Eder)

  → Exercise 5 (optimization)
- **Appendix**

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

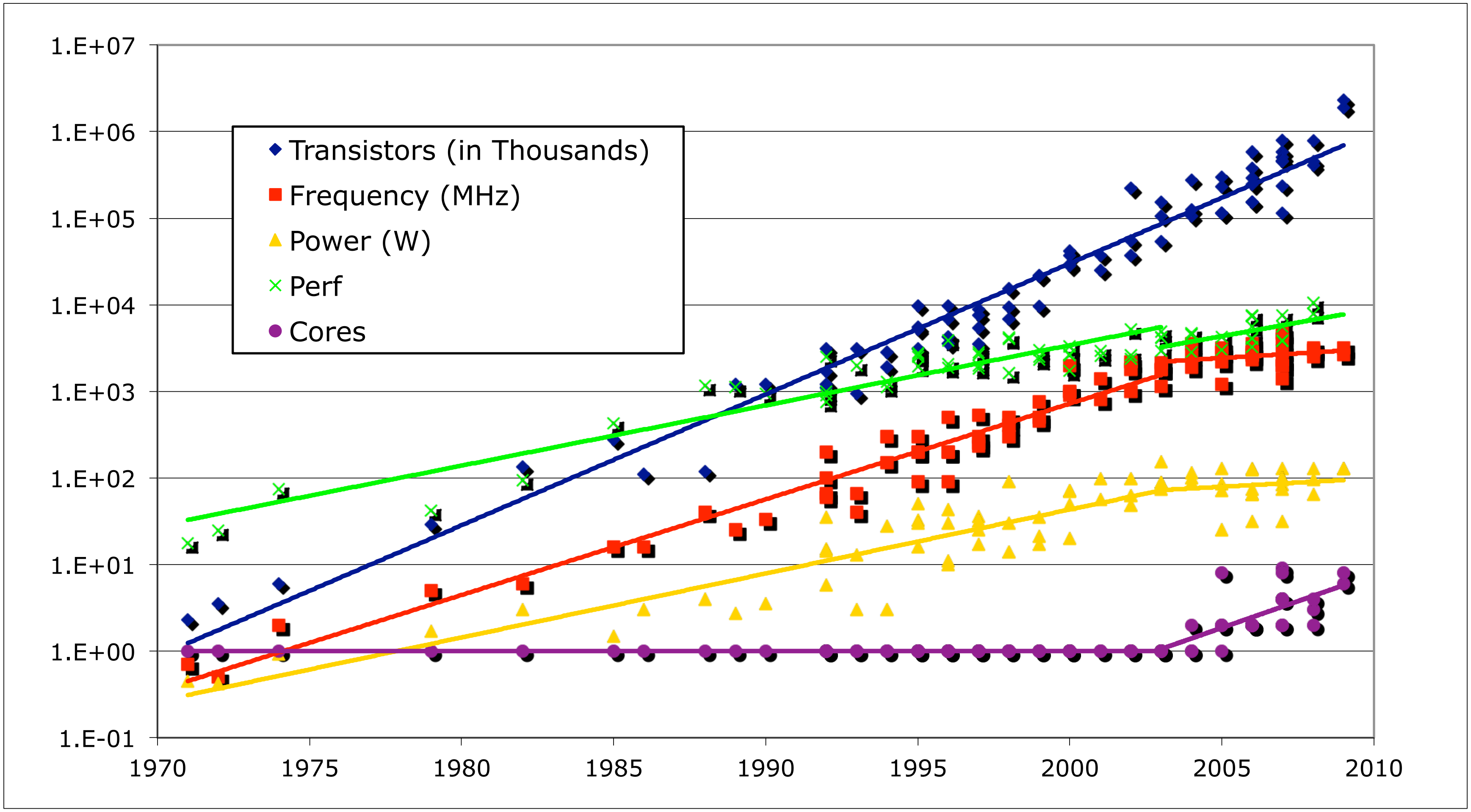
Basic PGAS Concepts

- Trends in hardware
- Execution model
- Memory model
- Run time environments
- Comparison with other paradigms
- Standardization efforts

Hands-on session: First UPC and CAF exercise

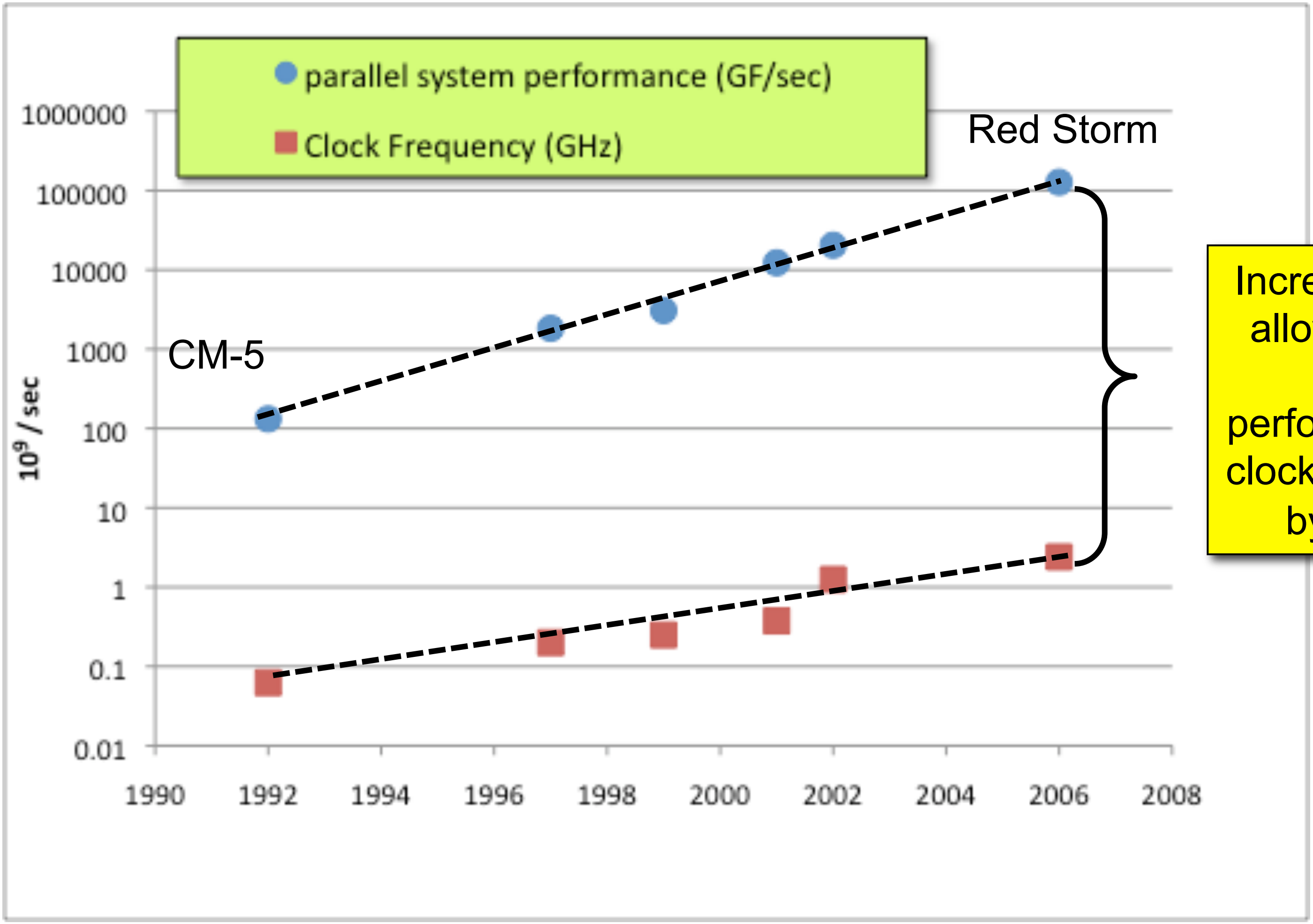
Moore's Law with Core Doubling Rather than Clock Speed

- ➢ Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Applications



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Concurrency was Part of the Performance Increase in the Past



Increased parallelism allowed a 1000-fold increase in performance while the clock speed increased by a factor of 40

and power, resiliency, programming models, memory bandwidth, I/O, …

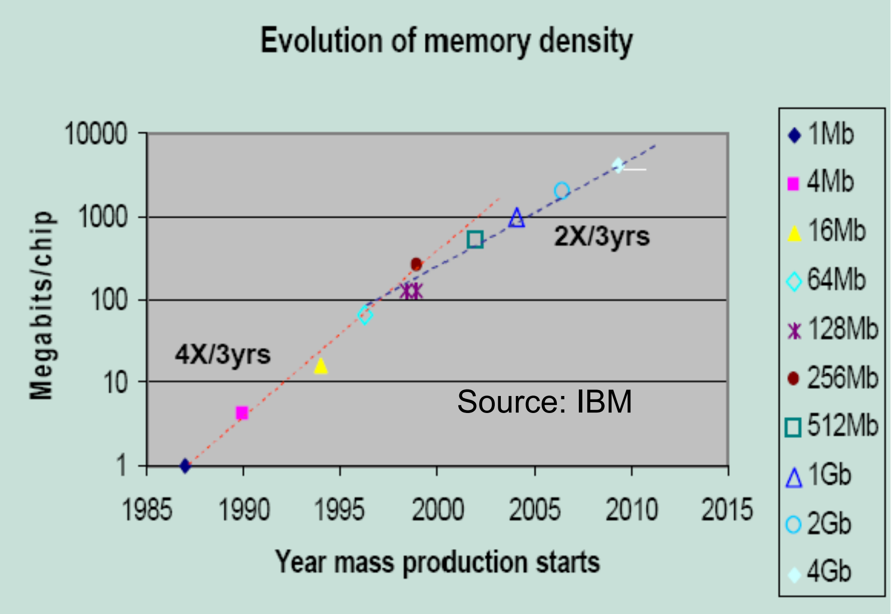
Exascale Initiative Steering Committee

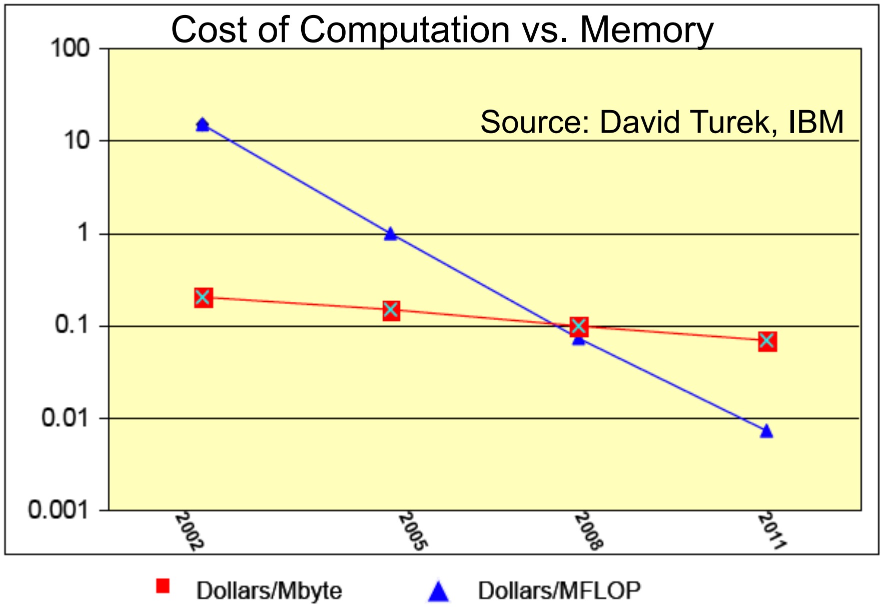
Memory is Not Keeping Pace

- ➢ Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Applications

Technology trends against a constant or increasing memory per core

- Memory density is doubling every three years; processor logic is every two
- Storage costs (dollars/Mbyte) are dropping gradually compared to logic costs

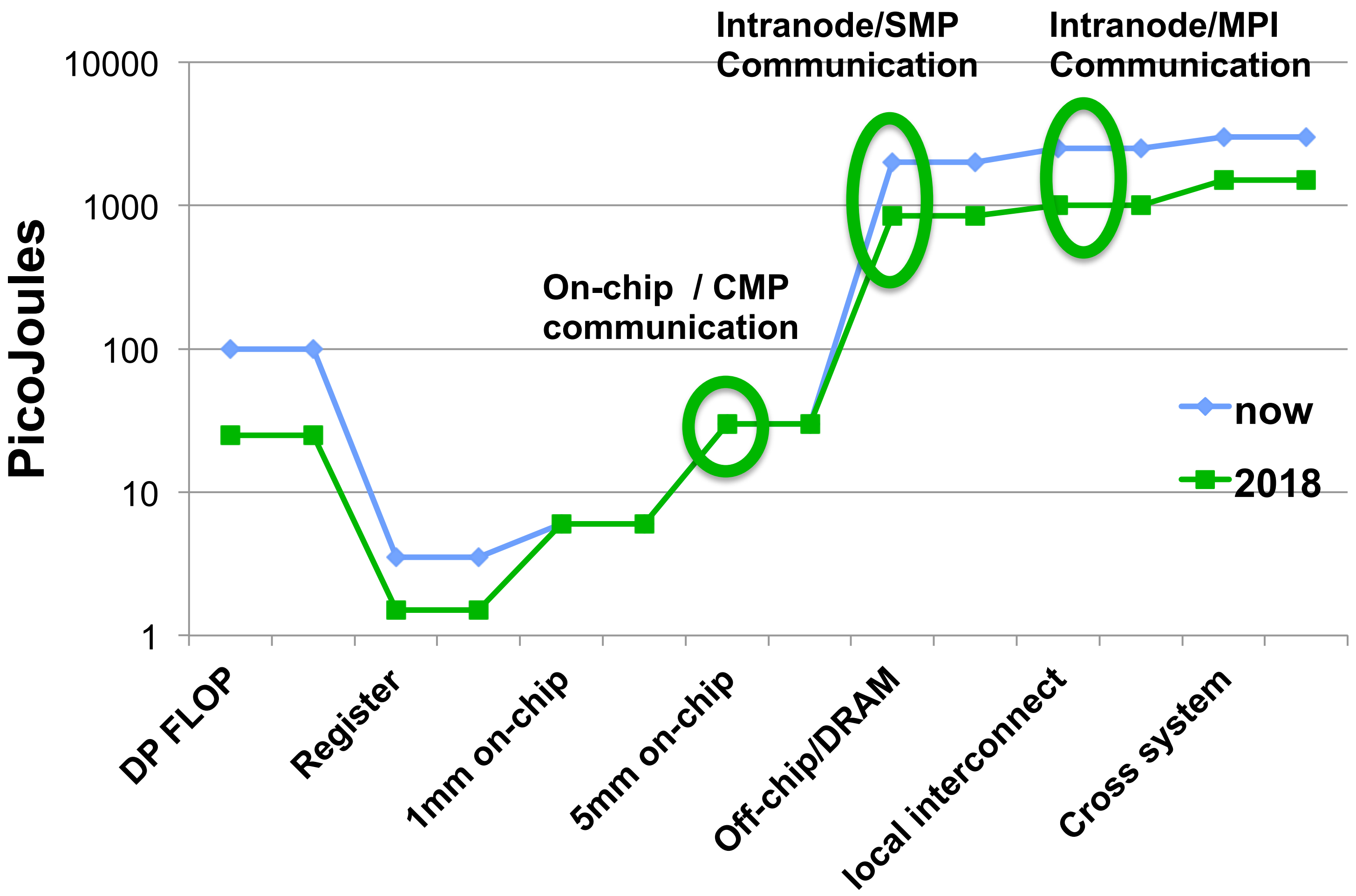




The cost to sense, collect, generate and calculate data is declining much faster than the cost to access, manage and store it

Question: *Can you double concurrency without doubling memory?*

Where the Energy Goes



Summary of Hardware Trends

- All future performance increases will be from concurrency
- Energy is the key challenge in improving performance
- Data movement is the most significant component of energy use
- Memory per floating point unit is shrinking

Programming model requirements

- Control over layout and locality to minimize data movement
- Ability to share memory to minimize footprint
- Massive fine and coarse-grained parallelism

Partitioned Global Address Space (PGAS) Languages

- ➢ **Basic PGAS concepts**
  - **Trends**
- **UPC and CAF basic syntax**
- **Advanced synchronization concepts**
- **Applications**

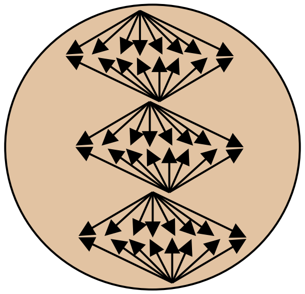
- **Coarray Fortran (CAF)**
  - Compilers from Cray, Rice and Intel (more soon)
- **Unified Parallel C (UPC)**
  - Compilers from Cray, HP, Berkeley/LBNL, Intrepid (gcc), IBM, SGI, MTU, and others
- **Titanium (Java based)**
  - Compiler from Berkeley

DARPA High Productivity Computer Systems (HPCS) language project:

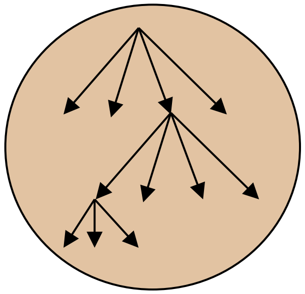
- **X10 (based on Java, IBM)**
- **Chapel (Cray)**
- **Fortress (SUN)**

Two Parallel Language Questions

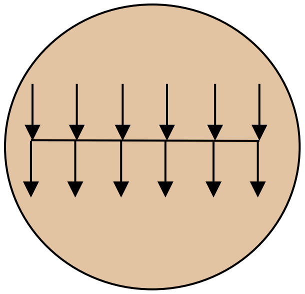
- What is the parallel control model?



data parallel
(single thread of control)

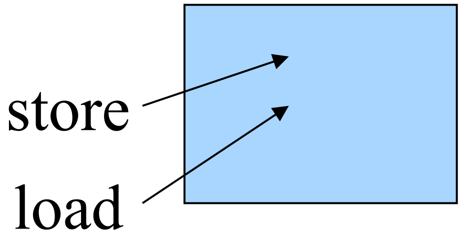


dynamic
threads

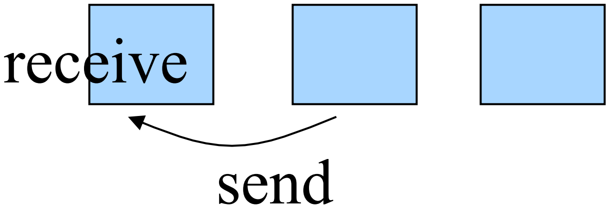


single program
multiple data (SPMD)

- What is the model for sharing/communication?



shared memory



message passing

implied synchronization for message passing, not shared memory

SPMD Execution Model

- **Single Program Multiple Data (SPMD) execution model**
  - Matches hardware resources: static number of threads for static number of cores ➔ no mapping problem for compiler/runtime
  - Intuitively, a copy of the main function on each processor
  - Similar to most MPI applications
- **A number of threads working independently in a SPMD fashion**
  - Number of threads given as program variable, e.g., `THREADS`
  - Another variable, e.g., `MYTHREAD` specifies thread index
  - There is some form of global synchronization, e.g., `upc_barrier`
  - Control flow (branches) are independent – not lock-step

- **UPC, CAF and Titanium: all use a SPMD model**
- **HPCS languages: X10, Chapel, and Fortress do not**
  - They support dynamic threading and data parallel constructs

Data Parallelism – HPF

```
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

- Data parallel languages use array operations (A = B, etc.) and loops
- Compiler and runtime map n-way parallelism to p cores
- Data layouts as in HPF can help with assignment using “owner computes”

- This mapping problem is one of the challenges in implementing HPF that does not occur with UPC and CAF

skipped

Dynamic Tasking - Cilk

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

The computation dag and parallelism unfold dynamically.

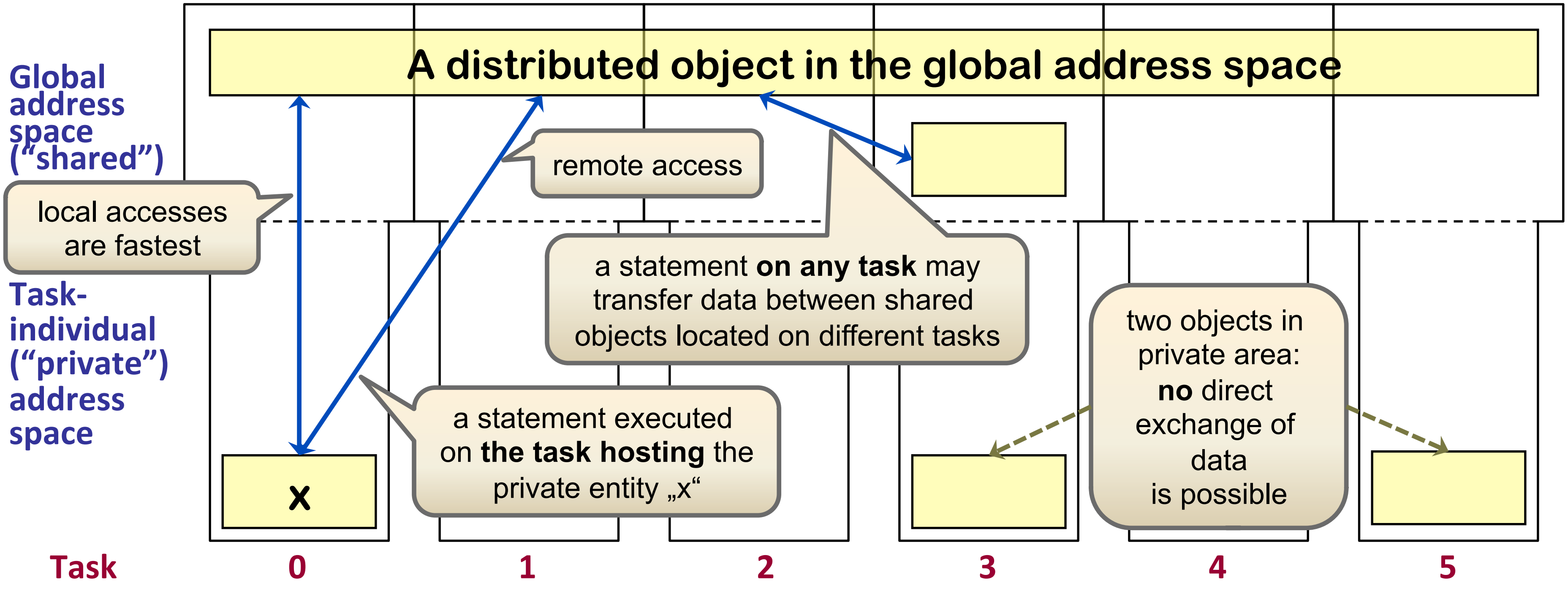
processors are virtualized; no explicit processor number

- **Task parallel languages are typically implemented with shared memory**
- **No explicit control over locality; runtime system will schedule related tasks nearby or on the same core**
- **The HPCS languages support these in a PGAS memory model which yields an interesting and challenging runtime problem**

Partitioned Global Address Space (PGAS) Languages

- ➢ Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Applications

- **Defining PGAS principle: extended memory model**
  1) The *Global Address Space:* a special memory area that allows any task to read or write memory anywhere in the system
  2) It is *Partitioned* to allow an efficient implementation of distributed objects (“symmetric heap”)

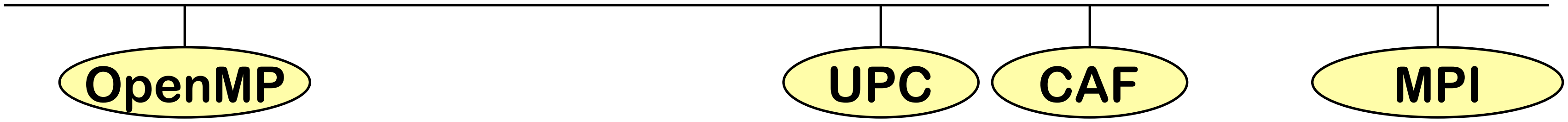

Global address space (“shared”)
A distributed object in the global address space
remote access
local accesses are fastest
a statement **on any task** may transfer data between shared objects located on different tasks
Task-individual (“private”) address space
x
a statement executed on **the task hosting** the private entity „x“
two objects in private area: **no** direct exchange of data is possible
Task
0
1
2
3
4
5

Two Concepts in the Memory Space

- **Private data: accessible only from a single thread**
  - Variable declared inside functions that live on the program stack are normally private to prevent them from disappearing unexpectedly
- **Shared data: data that is accessible from multiple threads**
  - Variables allocated dynamically in the program heap or statically at global scope may have this property
  - Some languages have both private and shared heaps or static variables
- **Local pointer or reference: refers to local data**
  - Local may be associated with a single thread or a shared memory node
- **Global pointer or reference (pointer-to-shared): may refer to "remote" data**
  - Remote may mean the data is off-thread or off-node
  - Global references are potentially remote; they *may* refer to local data

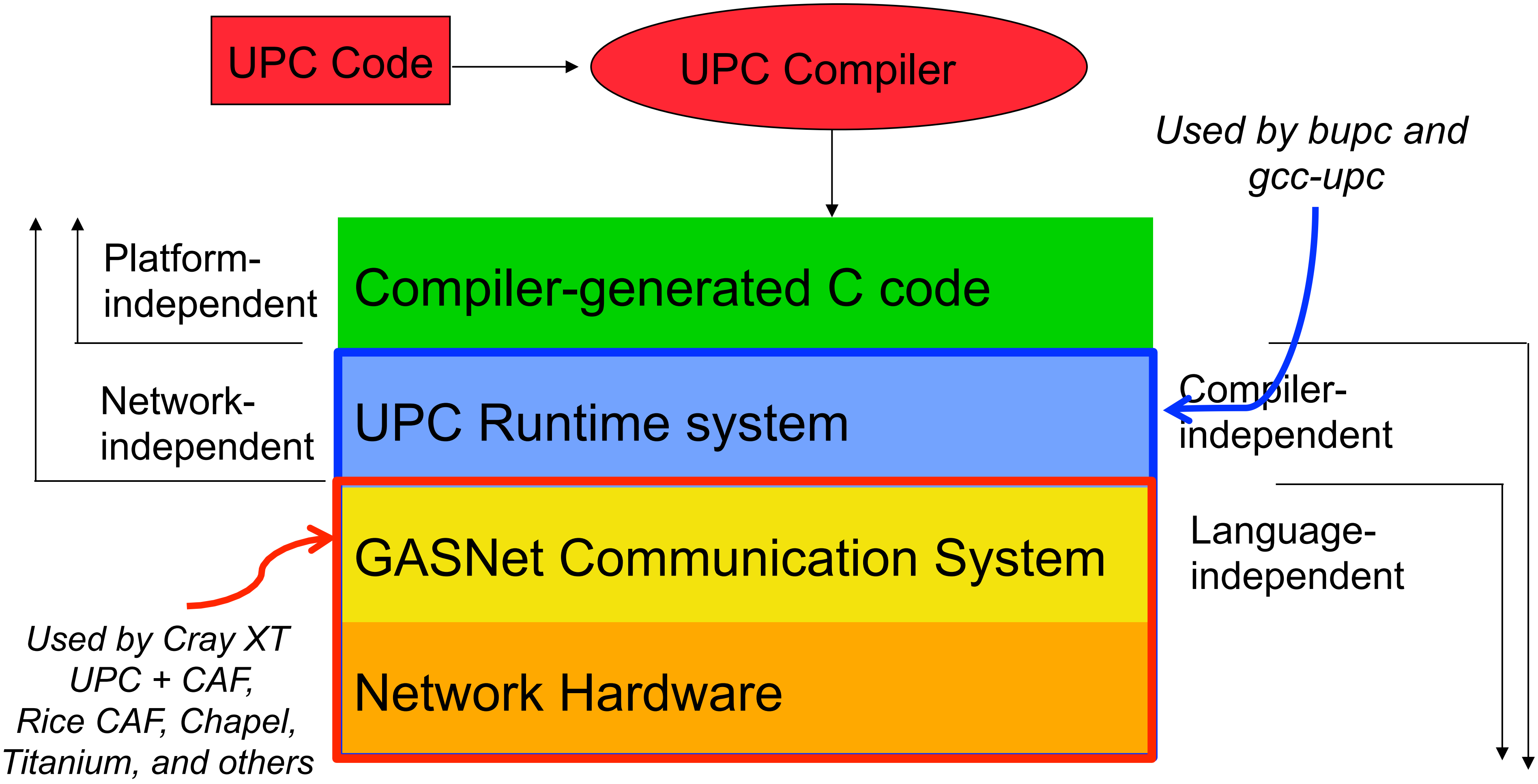
Other Programming Models

- **Message Passing Interface (MPI)**
  - Library with message passing routines
  - Unforced locality control through separate address spaces
- **OpenMP**
  - Language extensions with shared memory worksharing directives
  - Allows shared data structures without locality control



- **UPC / CAF data accesses:**
  - Similar to OpenMP but with locality control
- **UPC / CAF worksharing:**
  - Similar to MPI

Understanding Runtime Behavior - Berkeley UPC Compiler



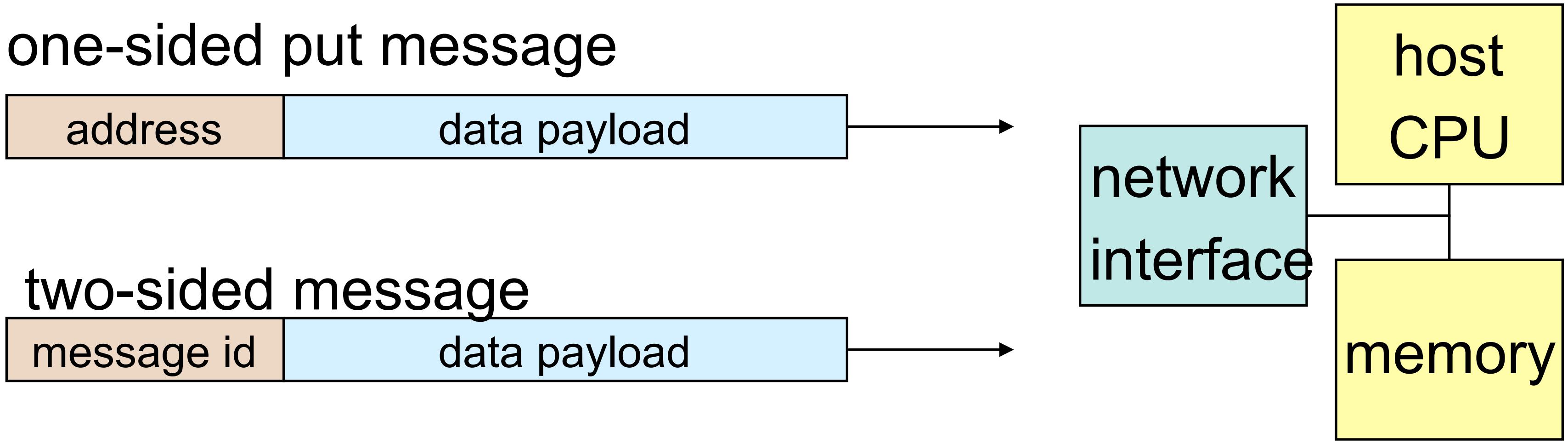
UPC Pointers

- **UPC pointers to shared objects have (conceptually) three fields:**
  - thread number
  - local address of block
  - phase (specifies position in the block) so that pointer arithmetic operations (like ++) move through the array correctly (more on blocks later)

- **Example implementation**

Phase	Thread	Virtual Address
63 49	48 38	37 0

One-Sided vs Two-Sided Communication



one-sided put message

address | data payload

two-sided message

message id | data payload

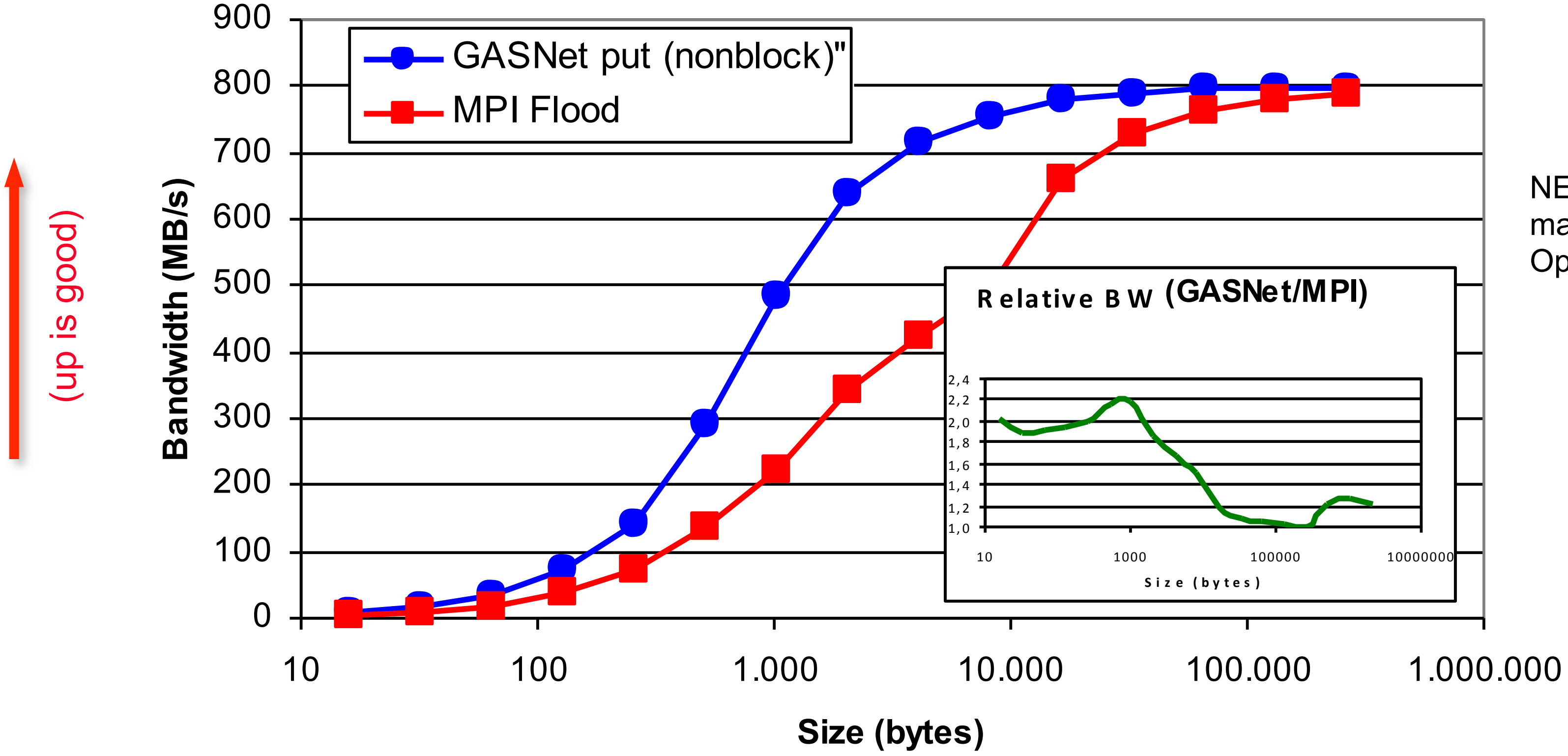
network interface

host CPU

memory

- **A one-sided put/get message can be handled directly by a network interface with RDMA support**
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- **A two-sided messages needs to be matched with a receive to identify memory address to put data**
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth

One-Sided vs. Two-Sided: Practice



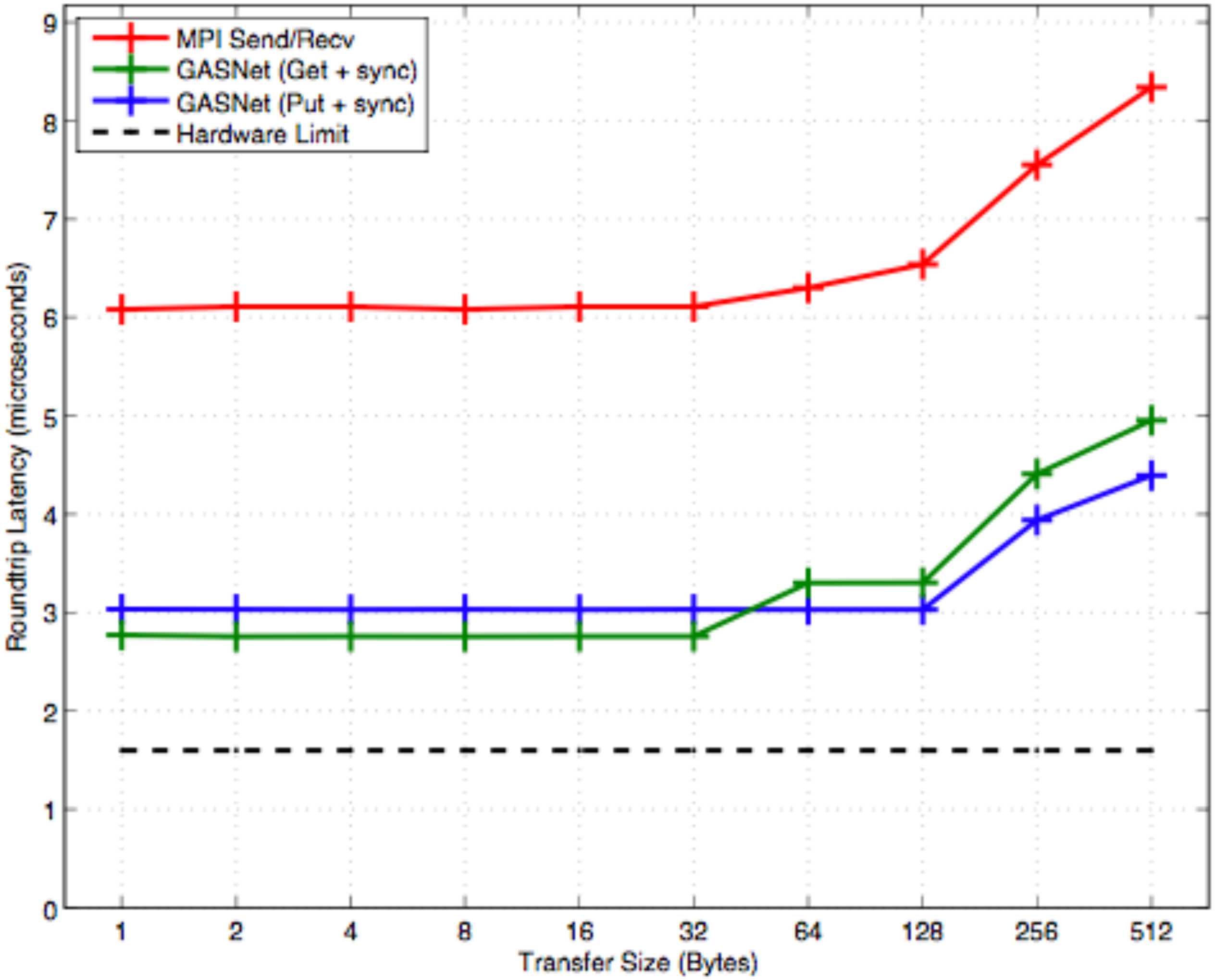
NERSC Jacquard machine with Opteron processors

- **InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5**
- **Half power point (N ½) differs by *one order of magnitude***
- **This is not a criticism of the implementation!**

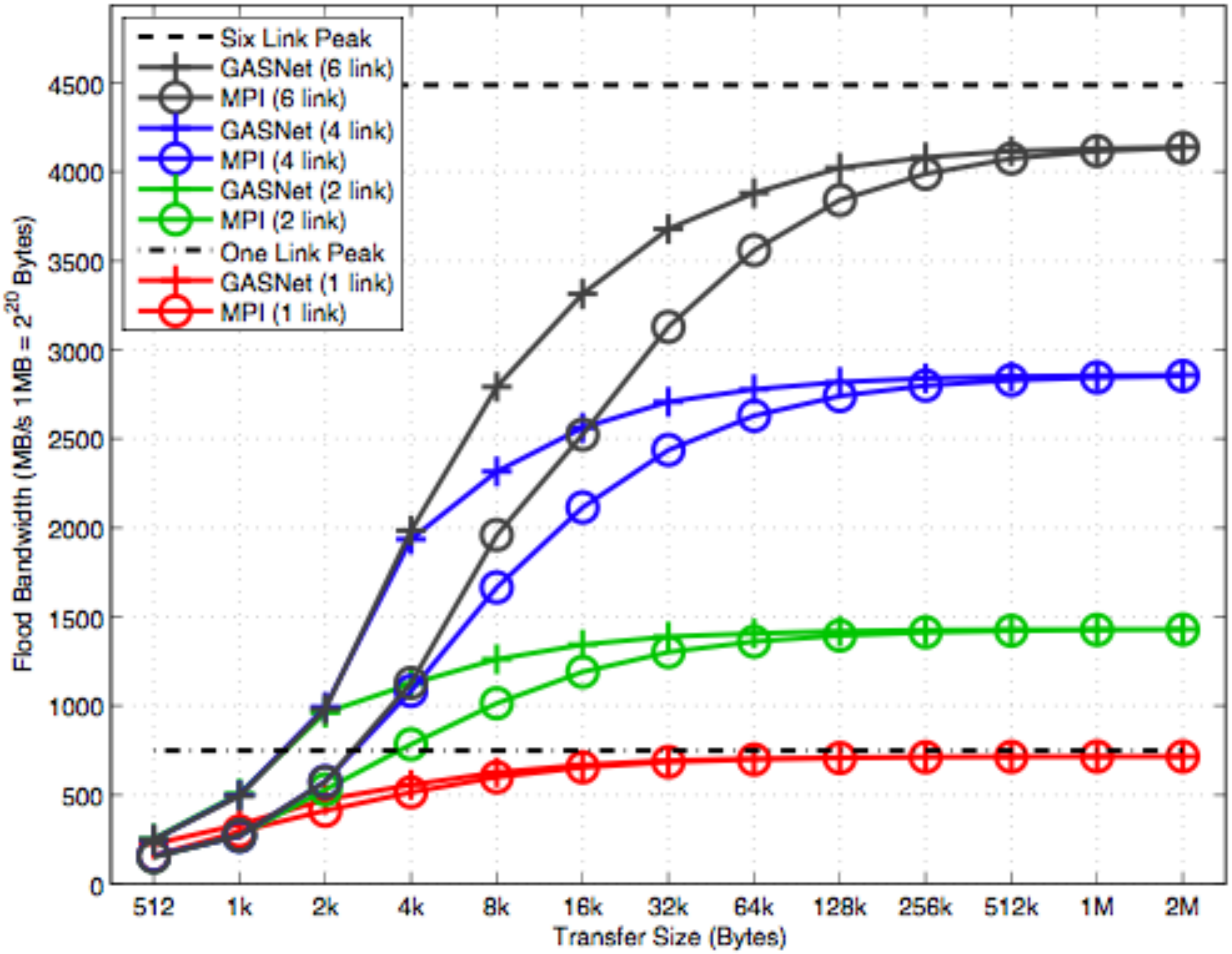
Joint work with Paul Hargrove and Dan Bonachea

GASNet vs MPI Latency on BG/P

- ➢ **Basic PGAS concepts**
  - **Trends**
- **UPC and CAF basic syntax**
- **Advanced synchronization concepts**
- **Applications**



GASNet vs. MPI Bandwidth on BG/P

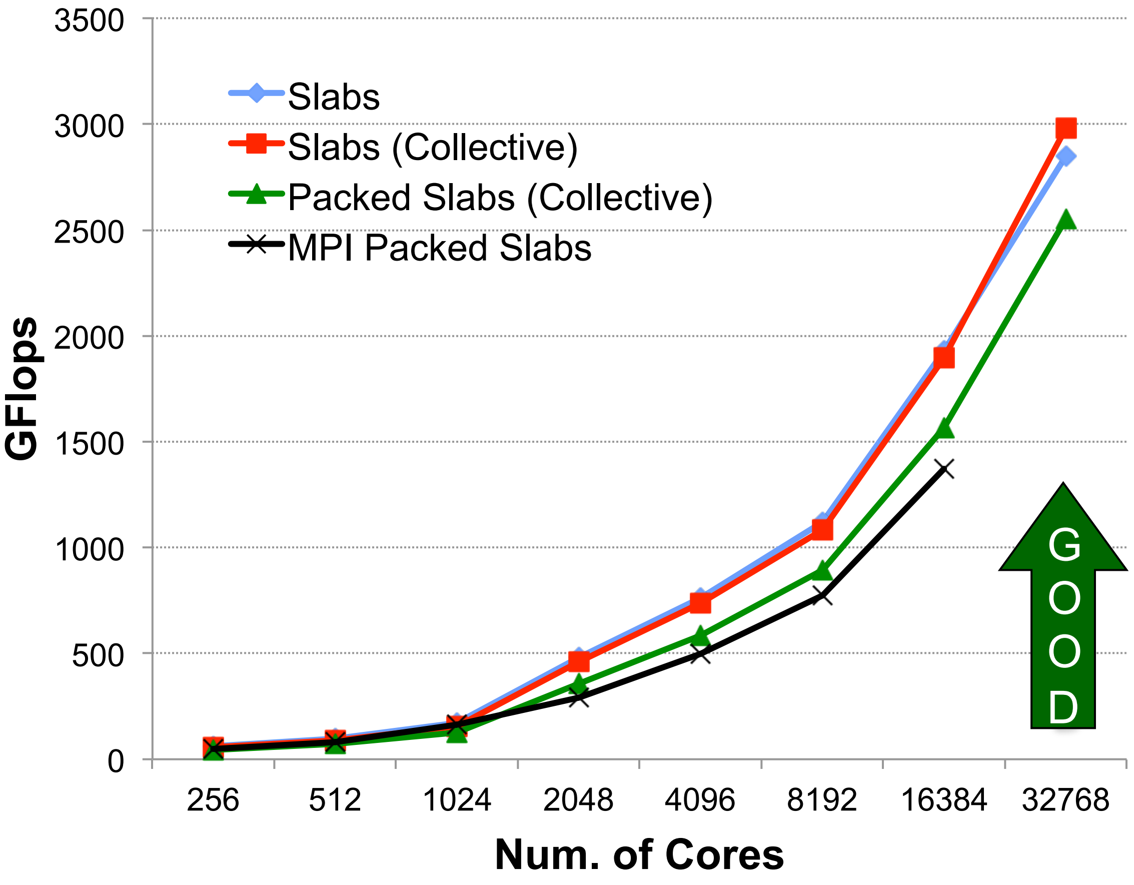


- **GASNet outperforms MPI on small to medium messages, especially when multiple links are used.**

FFT Performance on BlueGene/P

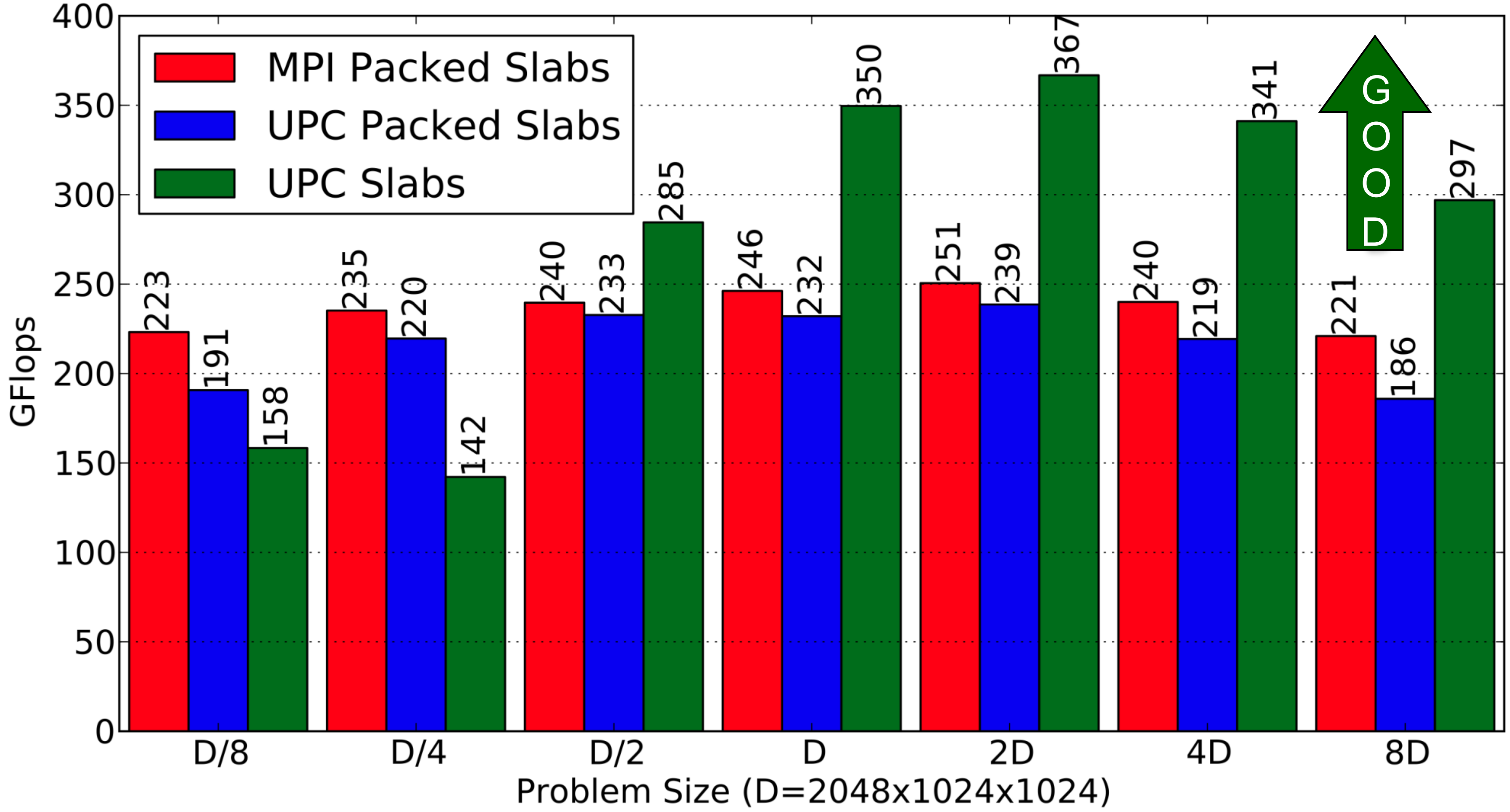
- PGAS implementations consistently outperform MPI
- Leveraging communication/computation overlap yields best performance
  - More collectives in flight and more communication leads to better performance
  - At 32k cores, overlap algorithms yield 17% improvement in overall application time
- Numbers are getting close to HPC record
  - Future work to try to beat the record

HPC Challenge Peak as of July 09 is ~4.5 Tflops on 128k Cores

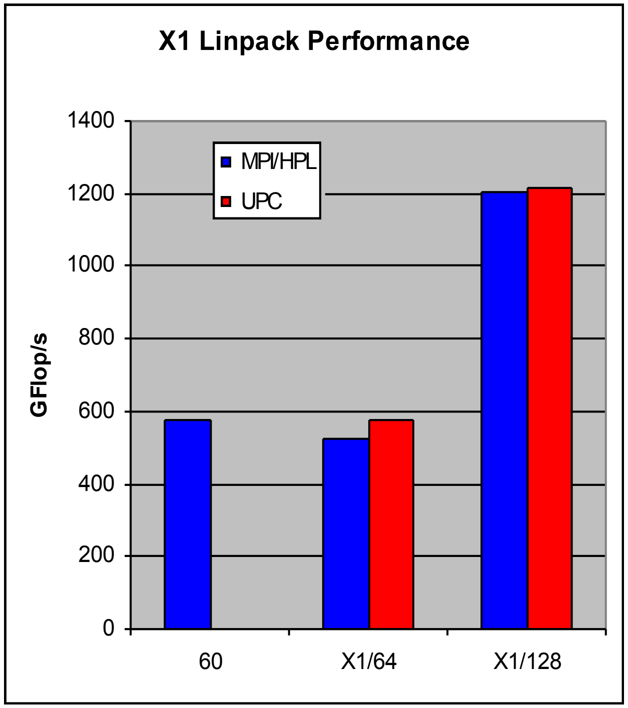


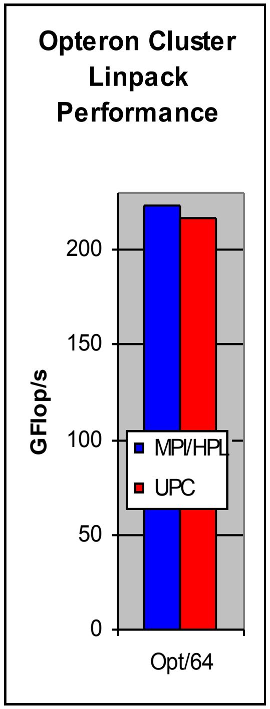
FFT Performance on Cray XT4

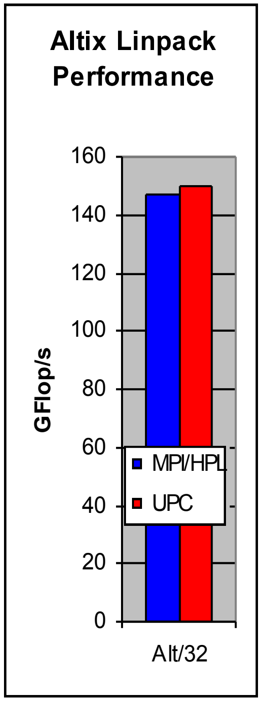
- **1024 Cores of the Cray XT4**
  - Uses FFTW for local FFTs
  - The larger the problem size the more effective the overlap



UPC HPL Performance


X1 Linpack Performance


Opteron Cluster Linpack Performance


Altix Linpack Performance

- MPI HPL numbers from HPCC database
- Large scaling:
  - 2.2 TFlops on 512p,
  - 4.4 TFlops on 1024p (Thunder)

- **Comparison to ScaLAPACK on an Altix, a 2 x 4 process grid**
  - ScaLAPACK (block size 64) 25.25 GFlop/s (tried several block sizes)
  - UPC LU (block size 256) - 33.60 GFlop/s, (block size 64) - 26.47 GFlop/s
- **n = 32000 on a 4x4 process grid**
  - ScaLAPACK - **43.34 GFlop/s** (block size = 64)
  - UPC - **70.26 GFlop/s** (block size = 200)

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/ (unmaintained)
  - Part of Fortran 2008
  - Cray and Intel compilers, gfortran in development
  - http://www.g95.org/coarray.shtml (unmaintained)

Future developments

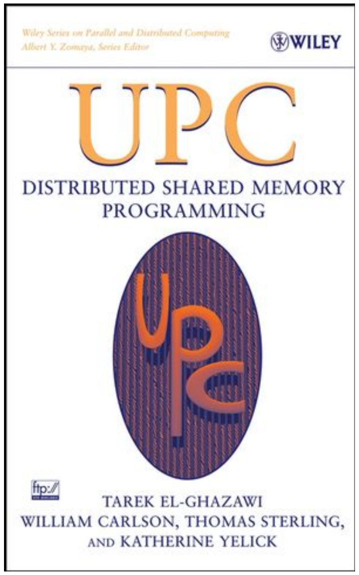
- **UPC**
  - Version 1.3 will define additional library functions
  - Feature List:
    - Improved support for lock deallocation, memory management, locality control
    - Non-blocking memory block transfers
    - Atomic Functions
- **CAF**
  - A Technical Specification has been proposed – if accepted, publication is targeted for 2014
    - TS to be integrated with next revision of the Fortran Standard
  - Feature List:
    - Collective Functions
    - Atomic Functions
    - One-sided synchronization (notify/query with events)
    - Composable Teams; includes a block construct that allows to define coarrays which only exist on subsets of images, and limits synchronization effects to the subset

will point out how new features fit into the concepts throughout this talk

skipped

UPC

- **UPC Language Specification (V 1.2)**
  - The UPC Consortium, June 2005
  - http://upc.gwu.edu/docs/upc_specs_1.2.pdf
- **UPC Manual**
  - Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, Tarek El-Ghazawi, May 2005
  - http://upc.gwu.edu/downloads/Manual-1.2.pdf
- **UPC Book**
  - Tarek El-Ghazawi, Bill Carlson, Thomas Sterling, and Katherine Yelick, June 2005

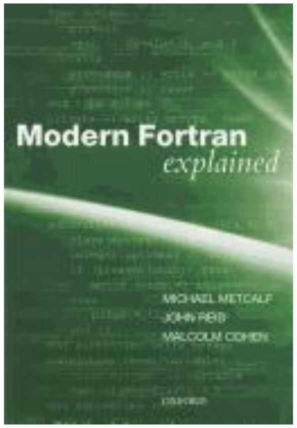


Wiley Series on Parallel and Distributed Computing
Albert Y. Zomaya, Series Editor
WILEY
UPC
DISTRIBUTED SHARED MEMORY PROGRAMMING
UPC
ftp://
TAREK EL-GHAZAWI
WILLIAM CARLSON, THOMAS STERLING,
AND KATHERINE YELICK

skipped

CAF

- On WG5 web site: John Reid:
  Co-arrays in the next Fortran Standard
  ISO/IEC JTC1/SC22/WG5 N1824 (April 21, 2010)
  - ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
- Metcalf, Reid and Cohen: **Modern Fortran Explained**
  OUP 2011, Chapter 19

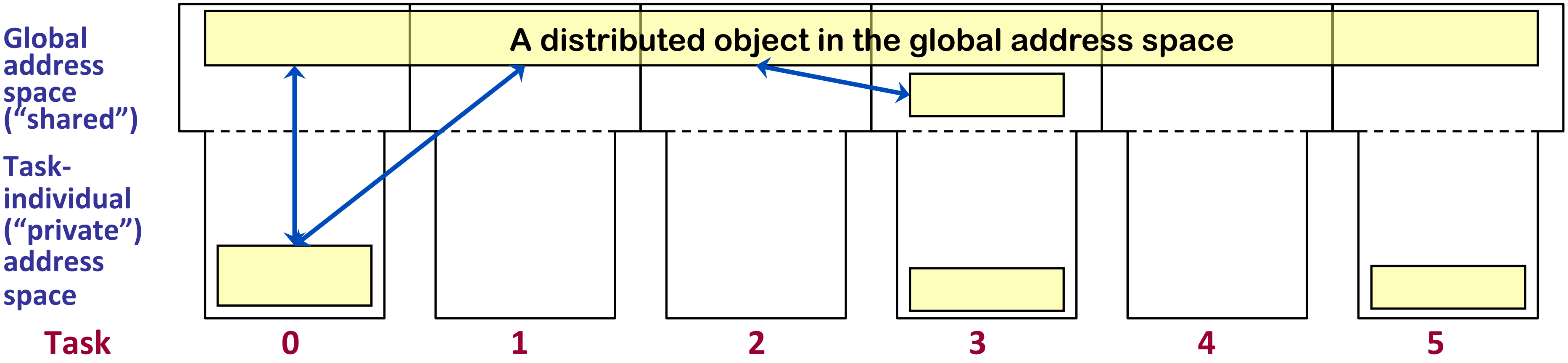


<u>Older papers:</u>

- 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

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's optimization 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


Global address space ("shared")
Task-individual ("private") address space
A distributed object in the global address space
Task
0
1
2
3
4
5

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
```

in original index range

remove range of lower processes

Data definition

Loop over y-dimension

Vectorizable loop over x-d

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

```
! 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
```

Irregular Applications

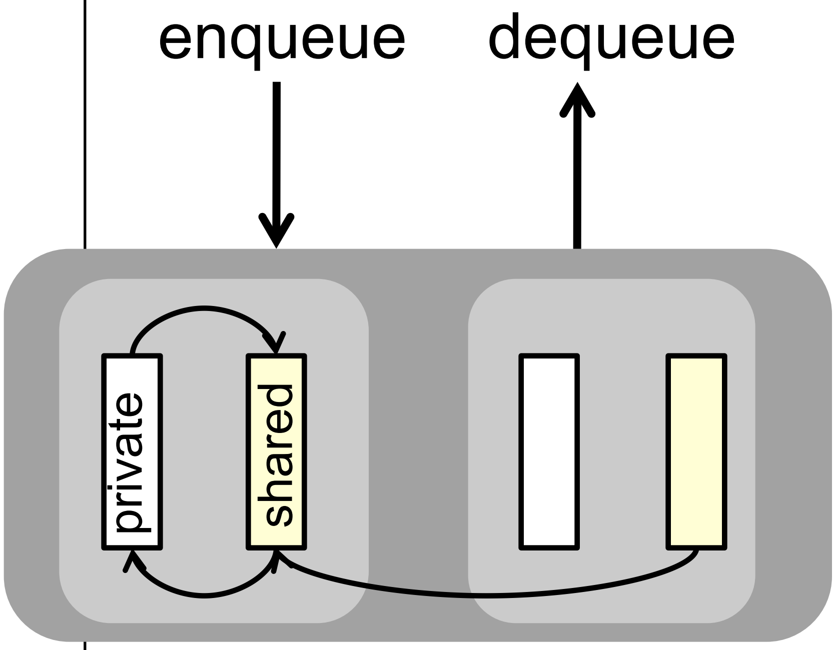
- **The SPMD model is too restrictive for some "irregular" applications**
  - The global address space handles irregular *data* accesses:
    - Irregular in space (graphs, sparse matrices, AMR, etc.)
    - Irregular in time (hash table lookup, etc.): for reads, UPC handles this well; for writes you need atomic operations
  - Irregular *computational* patterns are more difficult:
    - Not statically load balanced (even with graph partitioning, etc.)
    - Some kind of dynamic load balancing needed (e.g., a task queue)
- **Design considerations for dynamic scheduling UPC**
  - For locality reasons, SPMD still appears to be best for regular applications; aligns threads with memory hierarchy
  - UPC serves as "abstract machine model" so dynamic load balancing as an add-on may be written in portable UPC

Distributed Tasking API for UPC

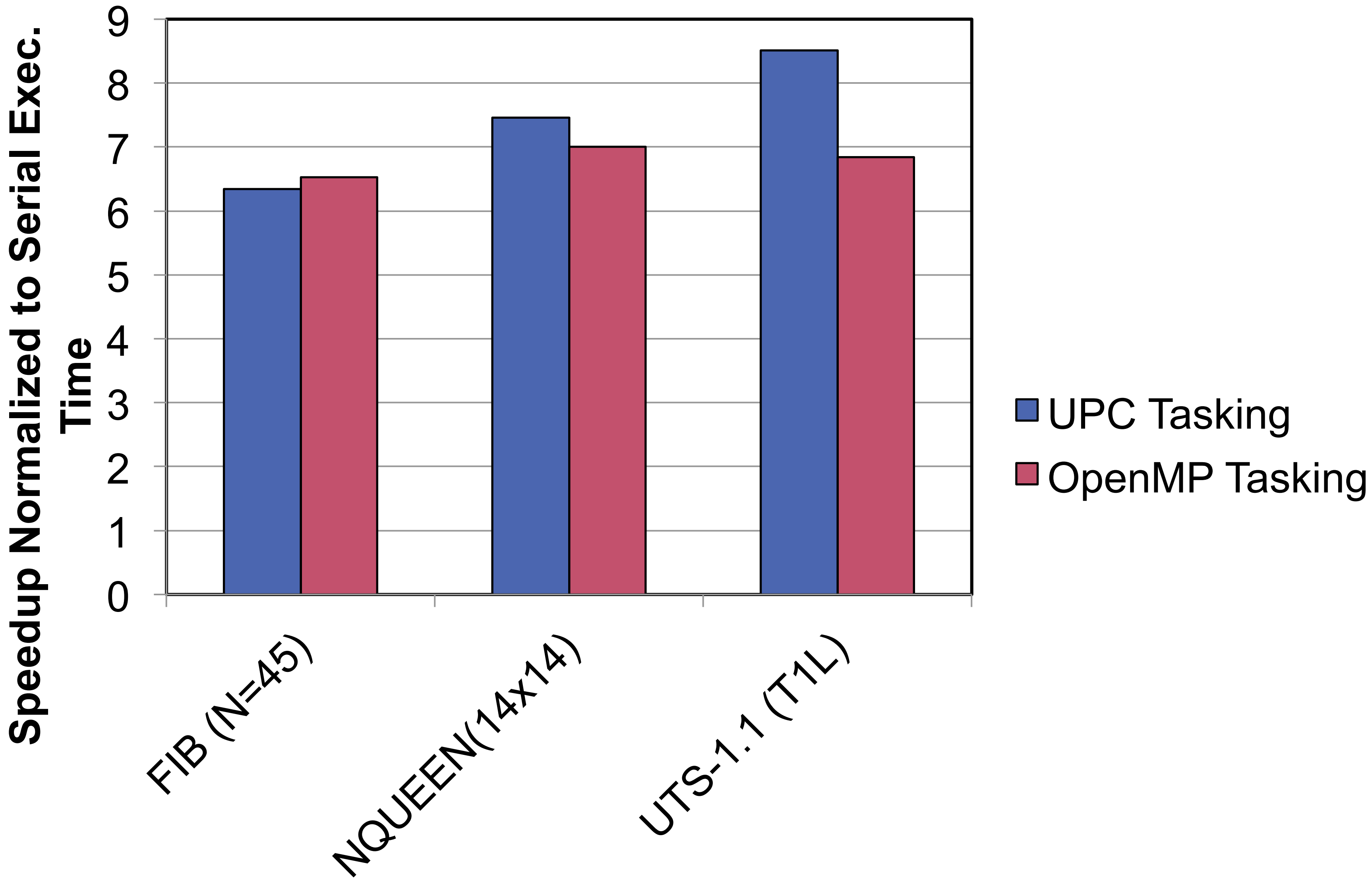
(http://upc.lbl.gov/task)

```
// allocate a distributed task queue
taskq_t * taskq_all_alloc();

// enqueue a task into the distributed queue
int taskq_put(taskq_t *, void *func,
                 void *in, void *out);

// run a task from the local task queue
// returns 0 if no task is available locally
int taskq_execute(taskq_t *);

// try to steal tasks from a random victim
int taskq_steal(taskq_t *);

// test whether queue is globally empty
int  taskq_isEmpty(taskq_t *);

// free distributed task queue memory
int  taskq_all_free(taskq_t *);
```

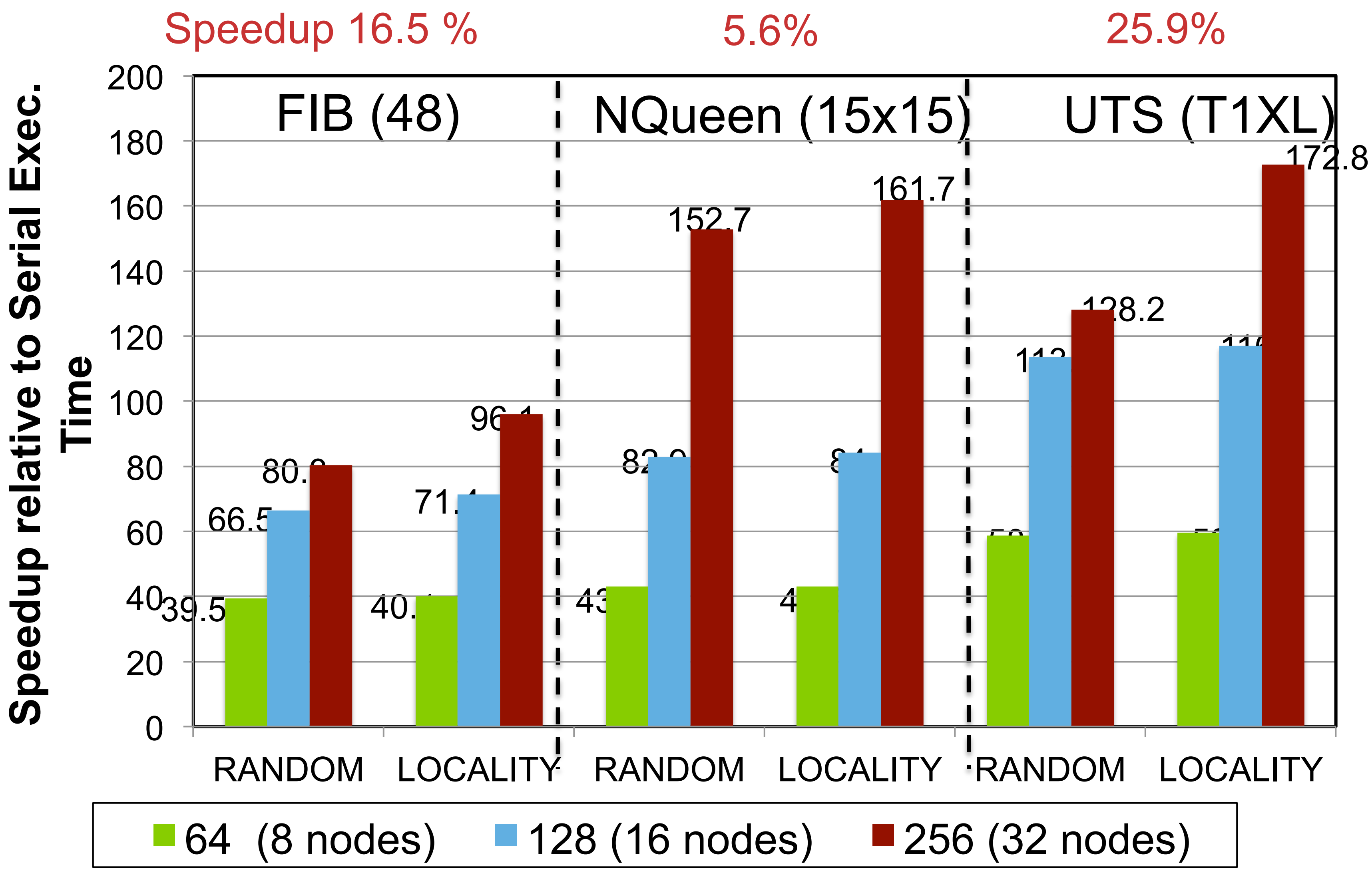
internals are hidden from user, except that dequeue operations may fail and provide hint to steal



UPC Tasking on Nehalem 8 core SMP

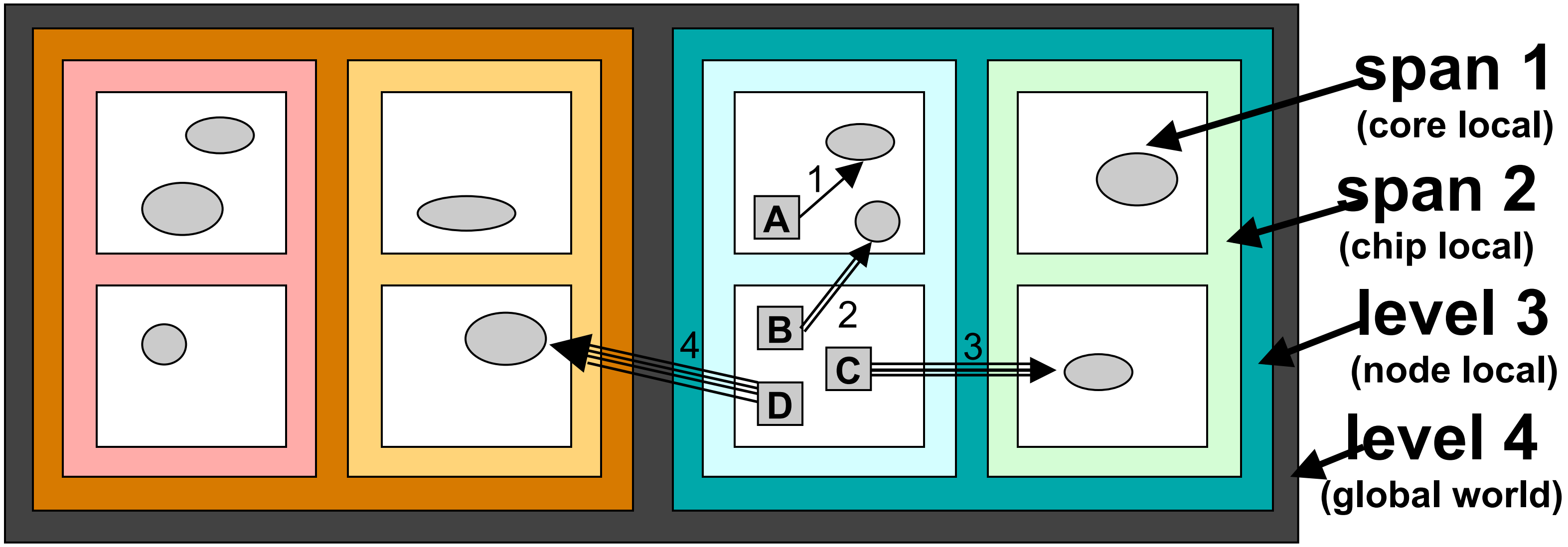


Multi-Core Cluster Performance



Hierarchical PGAS Model

- A global address space for hierarchical machines may have multiple kinds of pointers
- These can be encoded by programmers in type system or hidden, e.g., all global or only local/global
- This partitioning is about pointer span, not privacy control (although one may want to align with parallelism)



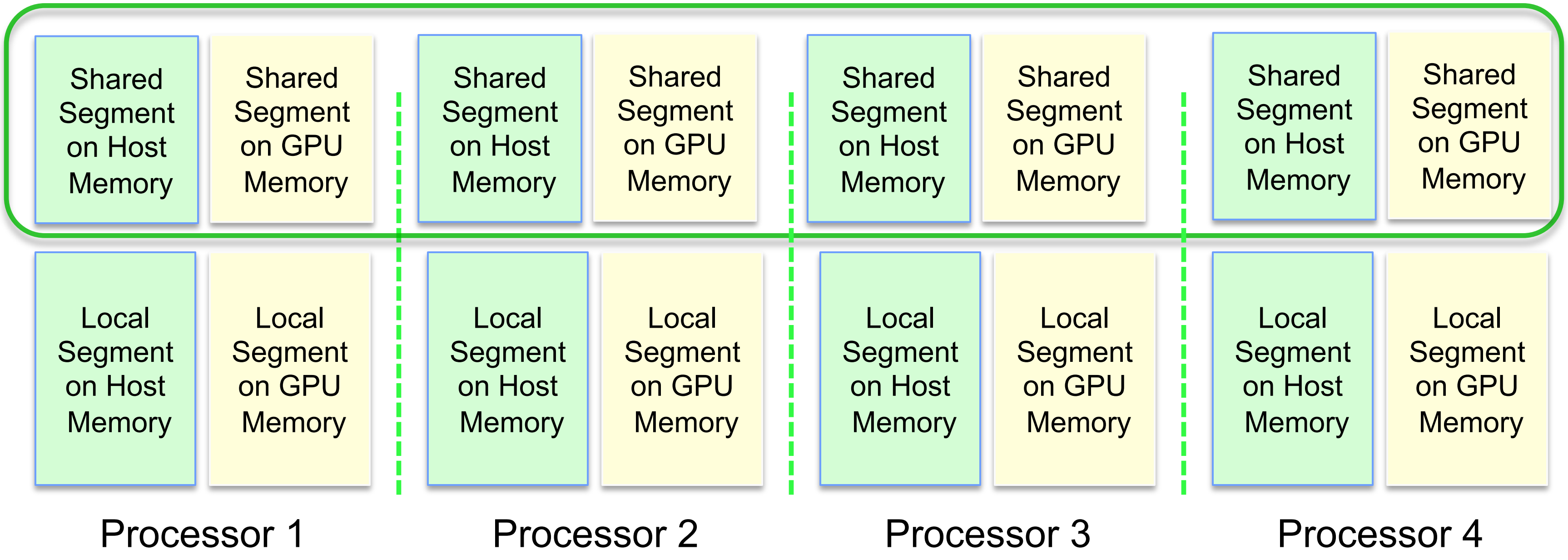
span 1
(core local)

span 2
(chip local)

level 3
(node local)

level 4
(global world)

Hybrid Partitioned Global Address Space

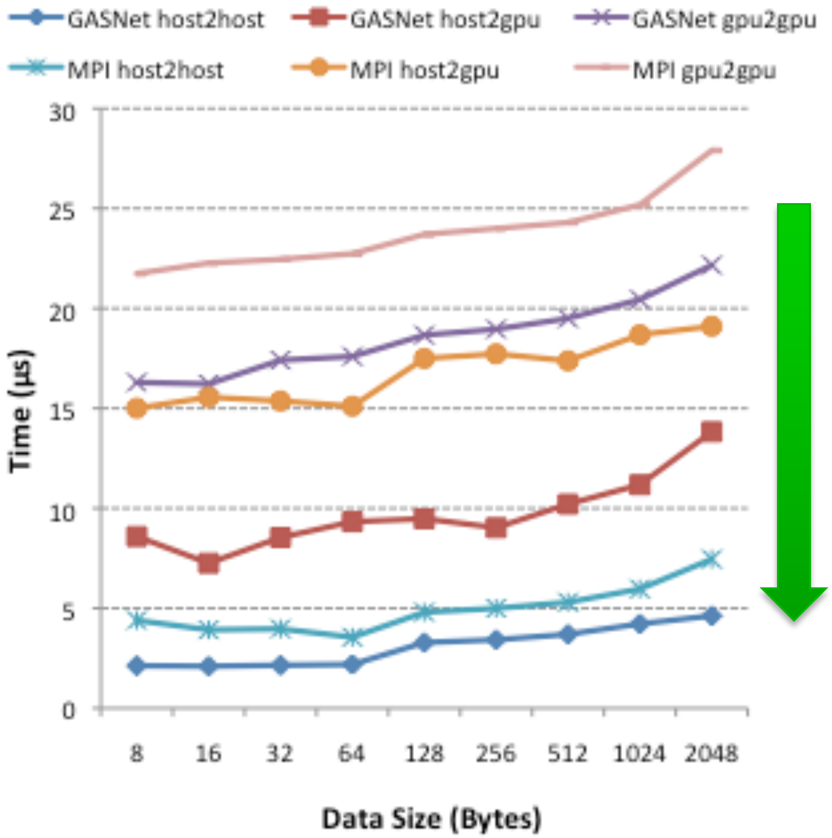


Processor 1		Processor 2		Processor 3		Processor 4	
Shared Segment on Host Memory	Shared Segment on GPU Memory	Shared Segment on Host Memory	Shared Segment on GPU Memory	Shared Segment on Host Memory	Shared Segment on GPU Memory	Shared Segment on Host Memory	Shared Segment on GPU Memory
Local Segment on Host Memory	Local Segment on GPU Memory	Local Segment on Host Memory	Local Segment on GPU Memory	Local Segment on Host Memory	Local Segment on GPU Memory	Local Segment on Host Memory	Local Segment on GPU Memory

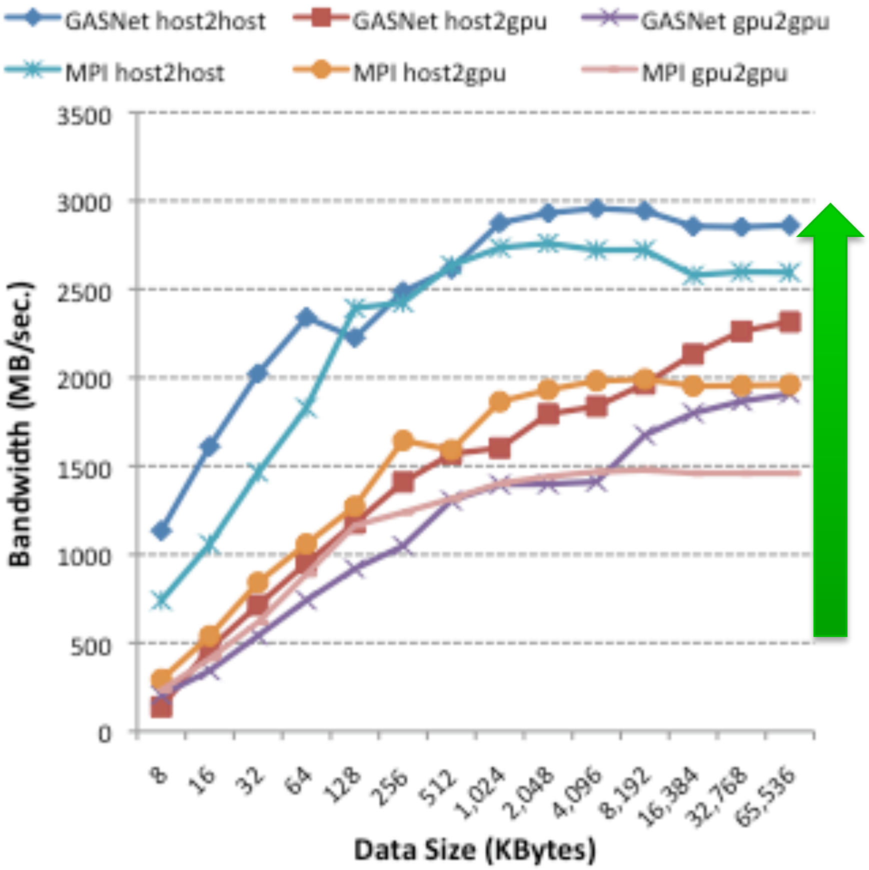
- ❖ Each thread has only two shared segments, which can be either in host memory or in GPU memory, but not both.
- ❖ Decouple the memory model from execution models; therefore it supports various execution models.
- ❖ Caveat: type system and therefore interfaces blow up with different parts of address space

GASNet GPU Extension Performance

Latency



Bandwidth



Compilation and Execution

- **On Cray XE6, hopper.nersc.gov (at NERSC), with PGI compiler**
  - **UPC only**
  - Initialization: `module load bupc`
  - Compile with fixed thread count:
    - `upcc -O -T=4 -o myprog myprog.c`
  - Compile with dynamic thread count:
    - `upcc -O -o myprog myprog.c`
  - Compile with debugging checks (assertions) enabled:
    - **`upcc -g [-T=4] -o myprog myprog.c`**
  - Execute (interactive test on 1 node with 24 cores):

```
qsub -I -q special -lmppwidth=24,mppnppn=24, \
                             walltime=00:30:00 -V
```

```
upcrun -n 4 -cpus-per-node 24 ./myprog
```

recommended for
any development work

Number of UPC threads:
Must equal the compile-time
„-T“ setting, if any

**see also
UPC-pgi**

Compilation and Execution

- **On Cray XE6, hopper.nersc.gov (at NERSC), with Cray compilers**
  - Initialization: `module switch PrgEnv-pgi PrgEnv-cray`
  - Compile:
    - UPC: `cc -h upc -o myprog myprog.c`
    - CAF: `ftn -e m -h caf -o myprog myprog.f90`
  - Execute (interactive test on 1 nodes with **24** cores):

    -
      ```
      qsub -I -q special -lmppwidth=24,mppnppn=24, \
                               walltime=00:30:00 -V
      ```
    - `aprun -n 24 -N 24 ./myprog` (all 24 cores in the node are used)
    - `aprun -n 12 -N 12 ./myprog` (only 12 cores are used)

see also Cray UPC

see also Cray Fortran

First exercise – part 1

- **Purpose:**
  - use of compiler and run time environment
  - use basic intrinsics
- **Copy skeleton program to your working directory:**
  - cp ../hello/hello_serial.f90   hello_caf_1.f90
  - cp ../hello/hello_serial.c   hello_upc_1.c
- **Add statements to enable running in parallel**
  - each task should write its rank and the number of tasks
  - only one task should write „hello world“
- **Compile and run**
  - with 4 tasks

hello

Fortran

```
integer :: myrank,numprocs
…
myrank   = this_image()
numprocs = num_images()
if (myrank == 1) &
    print *, 'Hello world'
print *,'I am image number', &
    myrank,' of ',numprocs
```

C

```
#include <upc.h>
…
if (MYTHREAD == 0)
    printf("hello world\n");
printf("I am thread number %d
  of %d threads\n",
  MYTHREAD, THREADS);
```

First exercise – part 2

- **Purpose:**
  - first attempt at data transfer
- **Copy program from part 1:**
  - cp hello_caf_1.f90 hello_caf_2.f90
  - cp hello_upc_1.c hello_upc_2.c
- **Add declaration for**
  - an integer coarray x (CAF)
  - an integer shared variable x (UPC)
- **Assign rank value on each task to x**
- **All tasks but the first should print the value of x on the first task**
  - observe what happens if run repeatedly with more than one image/thread

hello

Fortran

```
integer :: x[*] = 0
:
x = 99+this_image()
if (this_image() > 1) then
  write(*, *) ‘x from 1 on‘, &
      this_image(), ‘ is ‘,x[1]
end if
```

incorrect. why?

C

```
shared [*] int x[THREADS];
:
x[MYTHREAD] = 100+MYTHREAD;
if (MYTHREAD > 0) {
  printf(“x from 0 on %d is %d\n“
         , MYTHREAD, x[0]);
}
```

incorrect. why?

First exercise – part 3

- **Purpose:**
  - add necessary synchronization
- **Copy program from part 2:**
  - cp hello_caf_2.f90 hello_caf_3.f90
  - cp hello_upc_2.c hello_upc_3.c
- **Add synchronization statement**
- **Check correctness of results**

Fortran

```
integer :: x[*] = 0
:
x = 99+this_image()
sync all
if (this_image() > 1) then
  write(*, *) ‘x from 1 on‘, &
        this_image(), ‘ is ‘,x[1]
end if
```

C

```
shared [*] int x[THREADS];
:
x[MYTHREAD] = 100+MYTHREAD;
upc_barrier;
if (MYTHREAD > 0) {
  printf(“x from 0 on %d is %d\n“
              , MYTHREAD, x[0]);
}
```

hello

UPC and CAF Basic Syntax

- Declaration of shared data / coarrays
- Intrinsic procedures for handling shared data
  - elementary work sharing
- Synchronization:
  - motivation – race conditions;
  - rules for access to shared entities by different threads/images
- Dynamic entities and their management:
  - UPC pointers and allocation calls
  - CAF allocatable entities and dynamic type components
  - Object-based and object-oriented aspects

Hands-on: Exercises on basic syntax and dynamic data

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

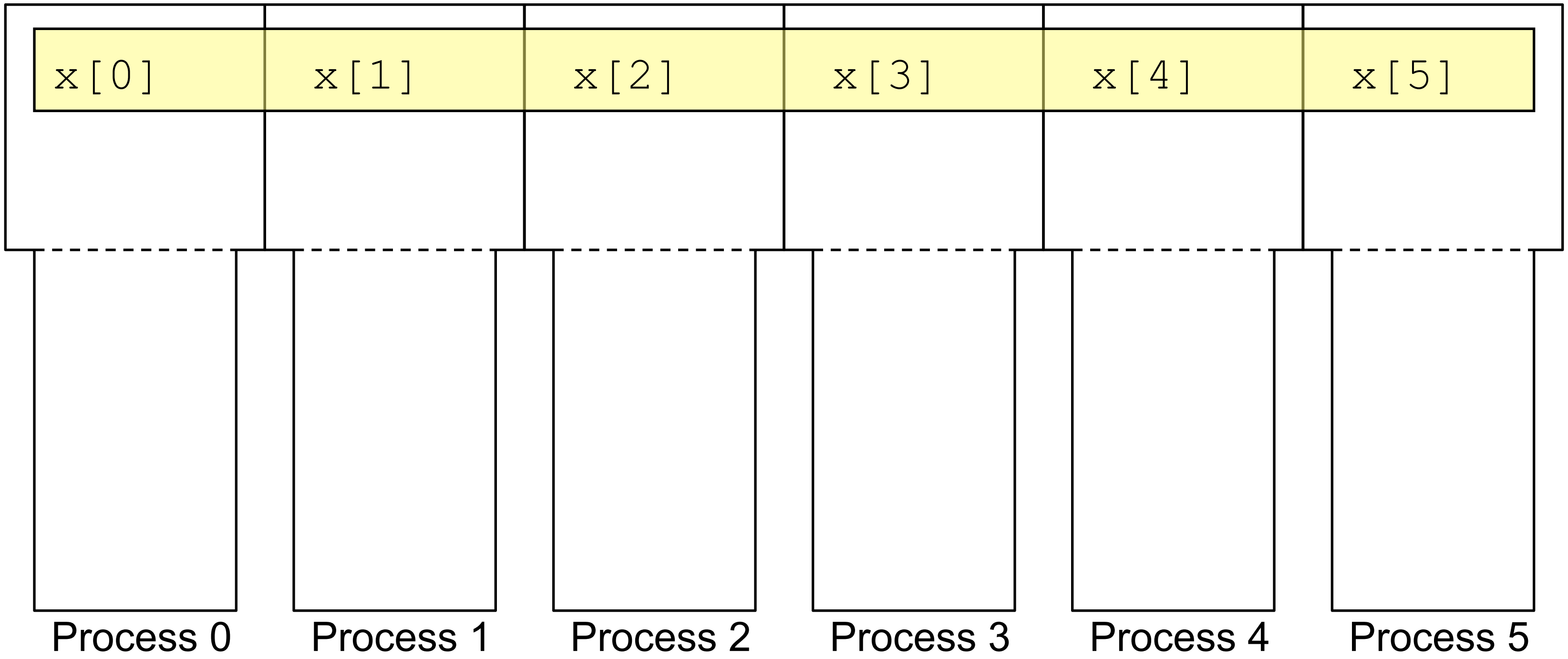
Partitioned Global Address Space: Distributed variable

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

UPC: “Parallel dimension”

CAF: “Codimension”

- **Data distribution:**



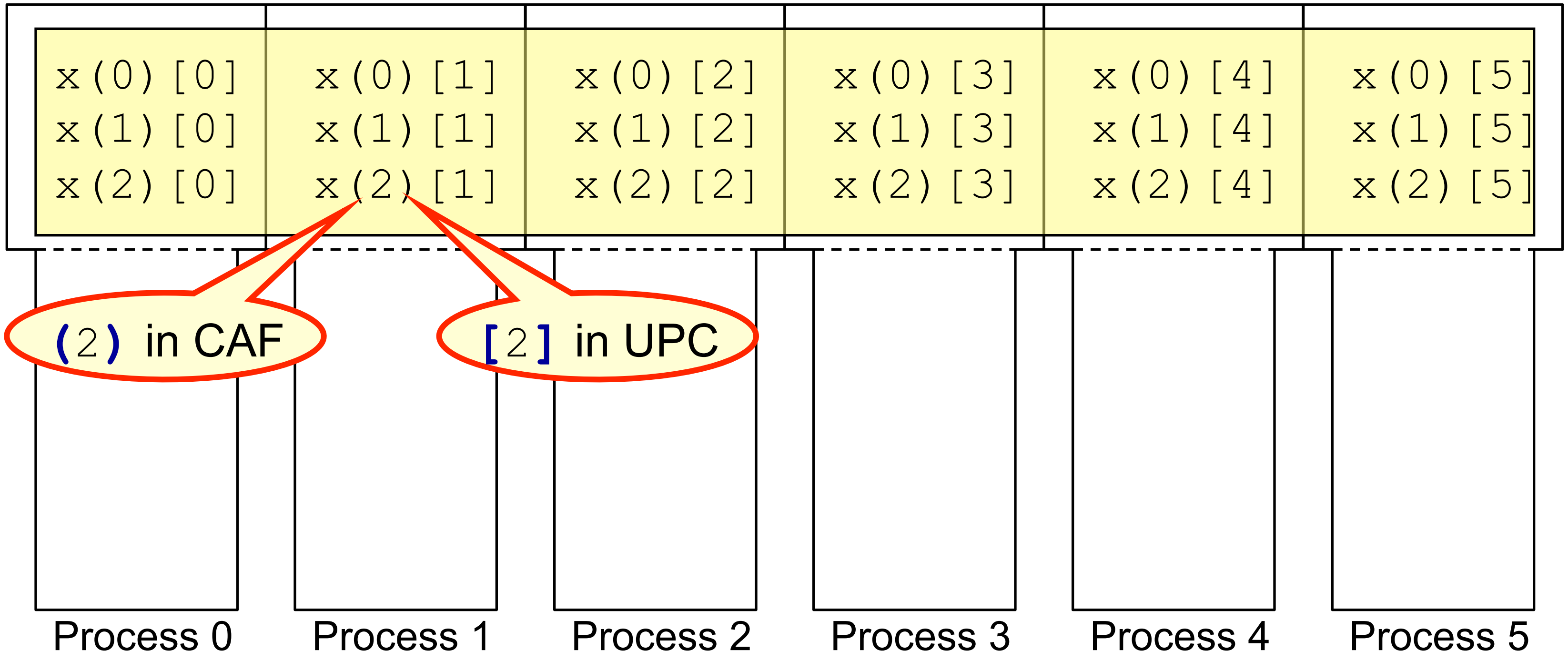
Partitioned Global Address Space: Distributed array

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

UPC: “Parallel dimension”

CAF: “Codimension”

- **Data distribution:**



Process 0	Process 1	Process 2	Process 3	Process 4	Process 5
x(0)[0]	x(0)[1]	x(0)[2]	x(0)[3]	x(0)[4]	x(0)[5]
x(1)[0]	x(1)[1]	x(1)[2]	x(1)[3]	x(1)[4]	x(1)[5]
x(2)[0]	x(2)[1]	x(2)[2]	x(2)[3]	x(2)[4]	x(2)[5]

(2) in CAF

[2] in UPC

Distributed arrays with UPC

- UPC shared objects may 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];
  - …

the dimensions define which elements exist

- Default: blocksize=1 if no "[**…**]" given (different from "[]", which we see later)
- The distribution is always round robin with chunks of **blocksize** elements
- Blocked distribution is implied if last dimension==THREADS and blocksize==1

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

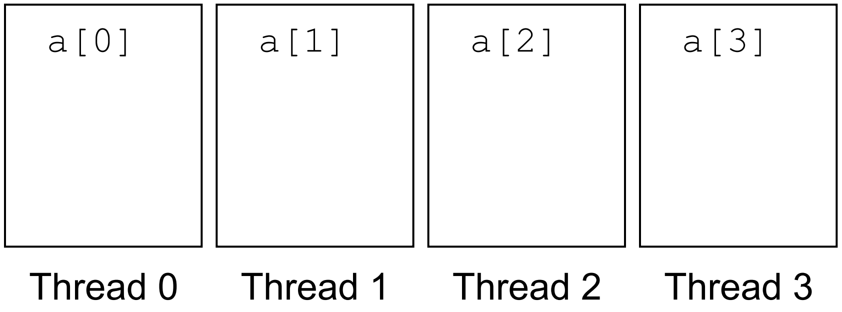
```
shared [5] float a[THREADS][5];
```

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]

Courtesy of Andrew Johnson

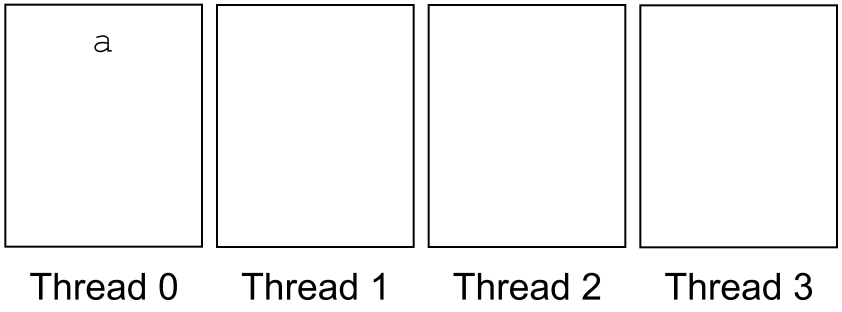
UPC shared data – examples (continued)

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



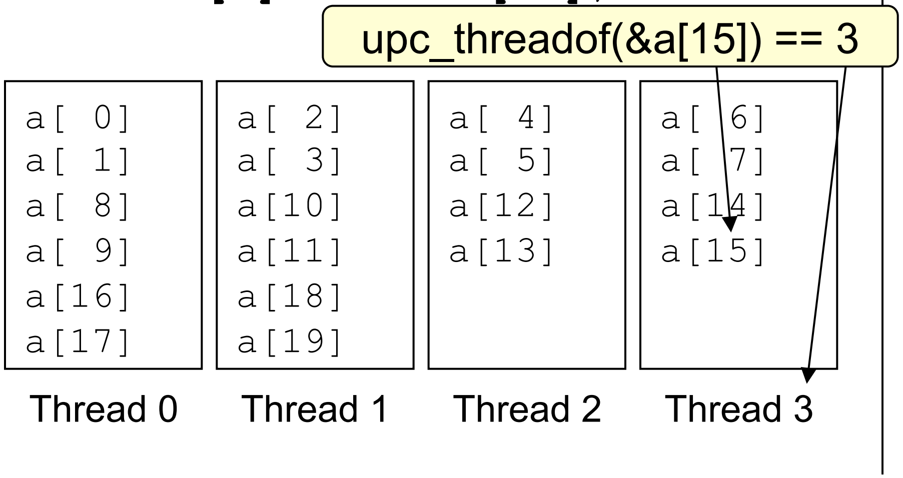
a[0]	a[1]	a[2]	a[3]
Thread 0	Thread 1	Thread 2	Thread 3

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



a			
Thread 0	Thread 1	Thread 2	Thread 3

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

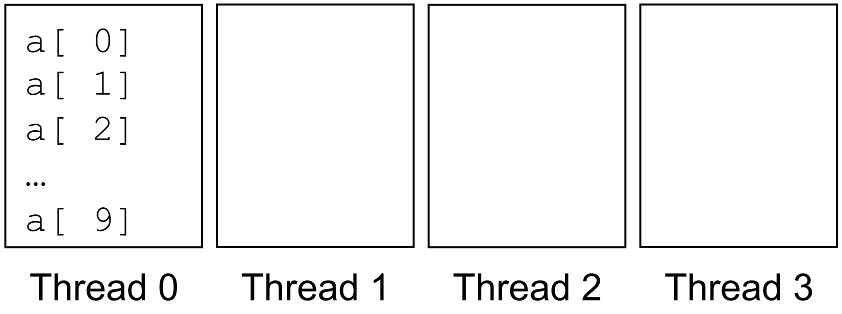


upc_threadof(&a[15]) == 3

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

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]			

Courtesy of Andrew Johnson

Integration of the type system
(static type components)

- **CAF:**

```
type :: body
  real :: mass
  real :: coor(3)
  real :: velocity(3)
end type
```

- **UPC:**

```
typedef struct {
  float mass;
  float coor[3];
  float velocity[3];
} Body;
```

declare and use entities of this type (symmetric variant):

```
type(body) :: asteroids(100)[*]
type(body) :: s
:
if (this_image() == 2) &
    s = asteroids(5)[1]
```

```
shared [100] \
   Body asteroids[THREADS][100];
Body s;
:
if (MYTHREAD == 1) {
  s = asteroids[0][4];
}
```

- compare this with effort needed to implement the same with MPI (dispense with **all** of `MPI_TYPE_*` API)
- what about dynamic type components? → later in this talk

Local access to local part of distributed variables (1)

- UPC:

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

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

Overrides compile-time type checking

  - `*x_local` now equals `x[MYTHREAD]`
  - can be used in its place for
    - clearer and more efficient code
    - passing data to standard (serial) numerical libraries
  - NOTE: generally, only allowed when datum x[i] has “local affinity”

```
upc_threadof(&x[i]) == MYTHREAD
```

  - FUTURE (UPC 1.3): equivalent for “intranode” sharing:

```
if (upc_castable(&x[i])) {
      x_local = upc_cast(&x[i]);
}
```

Enforces compile-time type checking

Local access to local part of distributed variables (2)

- **CAF: (0-based ranks)** **(1-based ranks)**

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

x = …
! x now equals x[myrank]
```

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

x = …
! x now equals x[myrank]
```

- **Most efficient way of accessing data**
  - For non-coindexed coarrays, it is **guaranteed** that no cross-image accesses occur
  - Therefore, the compiler can optimize code as if it were regular serial code

CAF-only: Multidimensional coindexing

- Coarrays may have a corank larger than 1
- Each variable may use a different coindex range

```
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)]
```

Remote access intrinsic support

- CAF: Inverse to `this_image()`: the `image_index()` intrinsic
  - delivers the image corresponding to a coindex tuple

```
integer :: remote_image
real :: y[0:1,0:*] ! high value of last coord must be *

remote_image = image_index(y, (/ 1, 2 /))
```

image on which y[1, 2] resides;
zero if coindex tuple is invalid

  - provides necessary information e.g., for future synchronization statements (to be discussed)

- UPC: `upc_threadof()` provides analogous information

Work sharing (1)

- **Loop execution**
  - simplest case: all data are generated locally

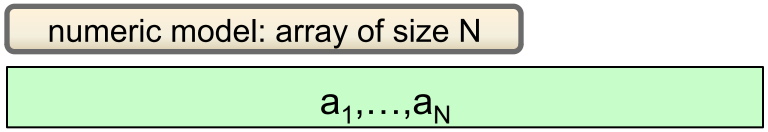
```
do i=1, n
  : ! do work
end do
```

  - chunking variants (`me=this_image()`)

```
do i=me,n,num_images()
  : ! do work
end do
```

```
: ! calculate chunk
do i=(me-1)*chunk+1,min(n,me*chunk)
  : ! do work
end do
```

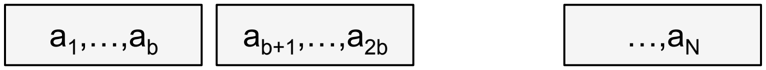
- **CAF data distribution**
  - in contrast to UPC, data model is fragmented
  - trade-off: performance vs. programming complexity



numeric model: array of size N

$a_1,\ldots,a_N$

  - blocked distribution:



$a_1,\ldots,a_b$ | $a_{b+1},\ldots,a_{2b}$ | $\ldots,a_N$

(block size: depends on number of images; number of actually used elements may vary between images)

  - alternatives: cyclic, block-cyclic

Work sharing (2)
data distribution + avoiding non-local accesses

- **CAF:**
  - index transformations between local and global

```
integer :: a(ndim)[*]
do i_local=1, nlocal
   i_global = …
   a(i_local) = …
end do
```

may vary between images

function of (i_local, this_image())

expression depends on i_global

- **UPC: global data model**
  - loop over all, work on subset

```
shared int a[N];
for (i=0; i<N; i++) {
  if (i%THREADS == MYTHREAD) {
     a[i] = … ;
  }
}
```

  - conditional may be inefficient
  - cyclic distribution may be slow

- **UPC: upc_forall**
  - integrates affinity with loop construct

```
shared int a[N];
upc_forall (i=0; i<N; i++; i) {
   a[i] = … ;
}
```

affinity expression

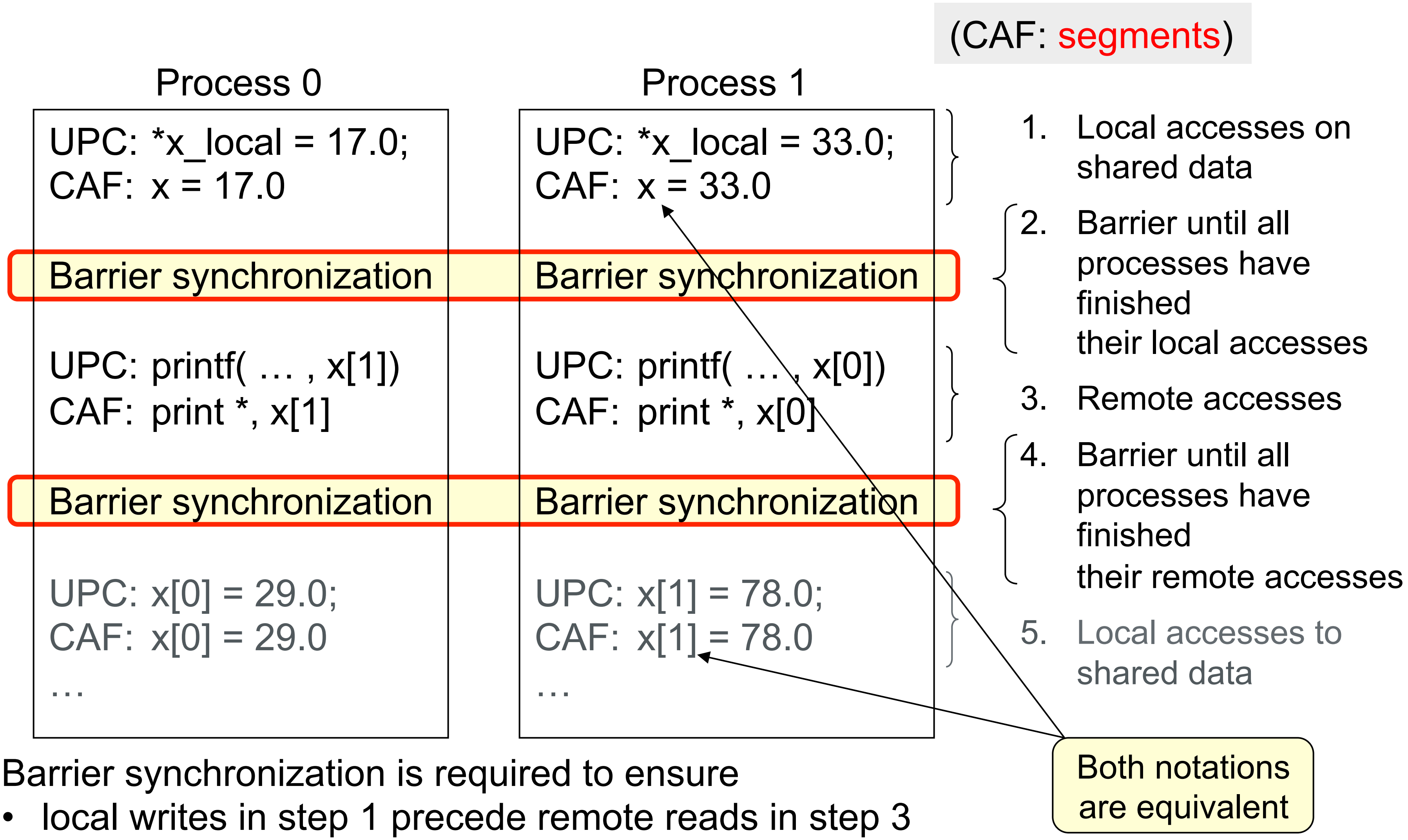
  - affinity expression:

    an integer → execute if `i %THREADS == MYTHREAD`

    a global address → execute if `upc_threadof(…) == MYTHREAD`

    `continue` or empty → all threads (use for nested upc_forall)
  - example above: could replace „i“ with „&a[i]“

Typical collective execution with access epochs (= synchronization phases)



(CAF: segments)

Process 0

UPC: *x_local = 17.0;
CAF: x = 17.0

Barrier synchronization

UPC: printf(… , x[1])
CAF: print *, x[1]

Barrier synchronization

UPC: x[0] = 29.0;
CAF: x[0] = 29.0
…

Process 1

UPC: *x_local = 33.0;
CAF: x = 33.0

Barrier synchronization

UPC: printf(… , x[0])
CAF: print *, x[0]

Barrier synchronization

UPC: x[1] = 78.0;
CAF: x[1] = 78.0
…

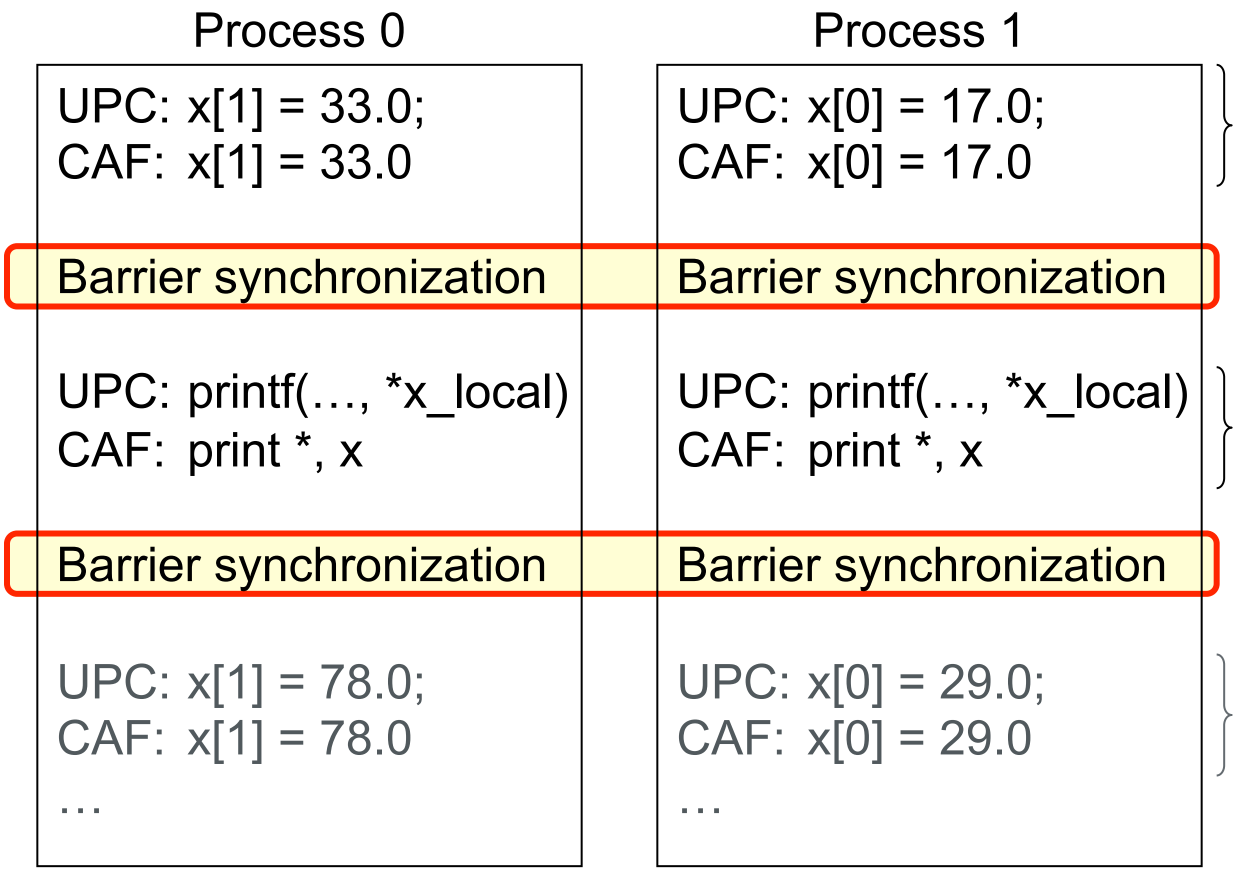
1. Local accesses on shared data
2. Barrier until all processes have finished
   their local accesses
3. Remote accesses
4. Barrier until all processes have finished
   their remote accesses
5. Local accesses to shared data

Both notations are equivalent

Barrier synchronization is required to ensure

- local writes in step 1 precede remote reads in step 3
- remote reads in step 3 precede local writes in step 5

Collective execution – same with remote write / local read



Process 0

UPC: x[1] = 33.0;
CAF: x[1] = 33.0

Barrier synchronization

UPC: printf(…, *x_local)
CAF: print *, x

Barrier synchronization

UPC: x[1] = 78.0;
CAF: x[1] = 78.0
…

Process 1

UPC: x[0] = 17.0;
CAF: x[0] = 17.0

Barrier synchronization

UPC: printf(…, *x_local)
CAF: print *, x

Barrier synchronization

UPC: x[0] = 29.0;
CAF: x[0] = 29.0
…

1. Remote accesses on shared data
2. Barrier until all processes have finished their remote accesses
3. Local accesses
4. Barrier until all processes have finished their local accesses
5. Remote accesses

Previous example with local/remote exchanged:

Barrier synchronization is required to ensure

- remote writes in step 1 precede local reads in step 3
- local reads in step 3 precede remote writes in step 5

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!

Otherwise race condition!

- 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
```

- Not the only synchronization mechanism, but the simplest one available

Examples

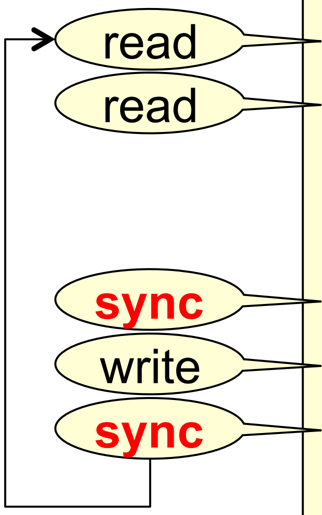
- UPC:

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

- CAF:

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

Another example



```
shared float x[THREADS];
while (!converged(x)) {
  float tmp = 0.5*x[MYTHREAD]
            - 0.25*x[(MYTHREAD+1)%THREADS]
            - 0.25*x[(MYTHREAD-1+THREADS)%THREADS];
  upc_barrier;
  x[MYTHREAD] = tmp;
  upc_barrier;
}
```

Note that real applications must do more work between synchronizations or performance will be horrible.

UPC and CAF Basic Syntax

- Declaration of shared data / coarrays
- Intrinsic procedures for handling shared data
  - elementary work sharing
- Synchronization:
  - motivation – race conditions;
  - rules for access to shared entities by different threads/images
- Dynamic entities and their management:
  - UPC pointers and allocation calls
  - CAF allocatable entities and dynamic type components
  - Object-based and object-oriented aspects

Hands-on: Exercises on basic syntax and dynamic data

Dynamic allocation with CAF

- **Coarrays may be allocatable:**

deferred shape/coshape

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

  – synchronization across all images is then implied at completion of the ALLOCATE statement (as well as at the start of DEALLOCATE)

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

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

- **Coarrays with POINTER attribute are not supported**

```
real,pointer :: ptr[*]   ! NOT supported: pointer coarray
```

  – this may change in the future

Dynamic entities: Pointers

- **Remember pointer semantics**
  - different between C and Fortran

Fortran

```
<type> , [dimension (:[,:,…])], pointer :: ptr

ptr => var          ! ptr is an alias for target var
```

no pointer arithmetic
type and rank matching

C

```
<type> *ptr;

ptr = &var;          ! ptr holds address of var
```

pointer arithmetic
rank irrelevant
pointer-to-pointer
pointer-to-void / recast

- **Pointers and PGAS memory categorization**
  - both pointer entity and pointee might be private or shared
    → **4 combinations** theoretically possible
  - **UPC: three** of these combinations are useful in practice
  - **CAF:** only **two** of the combinations allowed, and only in a limited manner
    ← aliasing is allowed only to local entities

Pointers continued …

- **CAF:**

```
integer, target :: i1[*]
integer, pointer :: p1

type :: ctr
  integer, pointer :: p2(:)
end type
type(ctr) :: o[*]
integer, target :: i2(3)
```

a coarray **cannot** have the `pointer` attribute

  – entity „o“: typically asymmetric

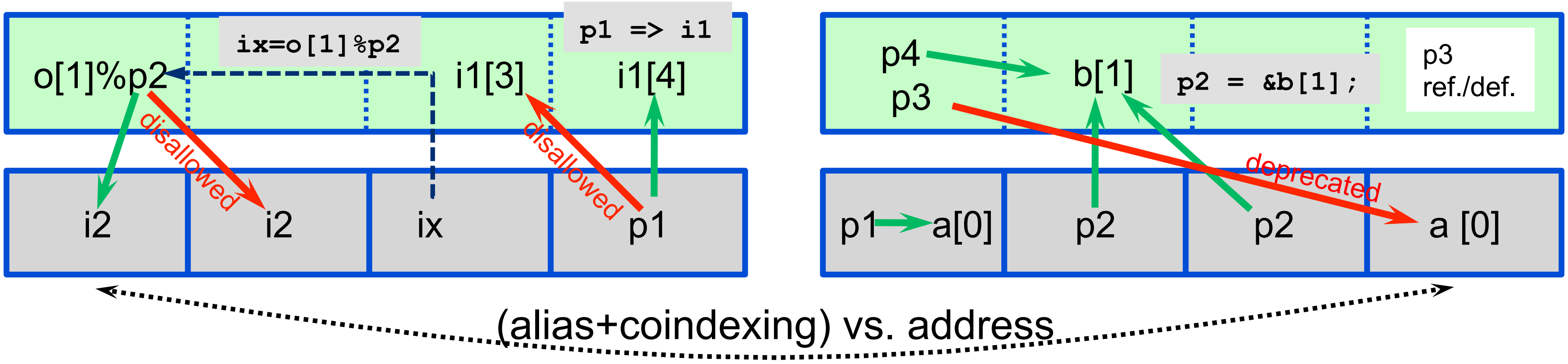
- **UPC:**

```
int *p1;
shared int *p2;
int *shared p3;
shared int *shared p4;
int a[N];
shared int b[N];
```

problem: where does p3 point?
all other threads may not reference

UPC: four combinations:
p1: **private** pointer to **private** memory
p2: **private** to **shared**
p3: **shared** to **private**
p4: **shared** to **shared**

  – pointer to shared: addressing overhead



Pointer to local portions of shared data (review)

- Cast a shared entity to a local pointer

```
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]
```

address must have affinity to **local** thread

pointer arithmetic selects local part

- May have performance advantages
- May improve code readability
- Required when passing to non-UPC numerical libraries

- Breaking the local-affinity rule (e.g., using a_local[5]) results in undefined behavior

UPC: to-shared Pointer blocking and casting

- **Assume 4 threads:**

```
shared [2] int A[10];
shared int *p2;
shared [2] int *q2;
```

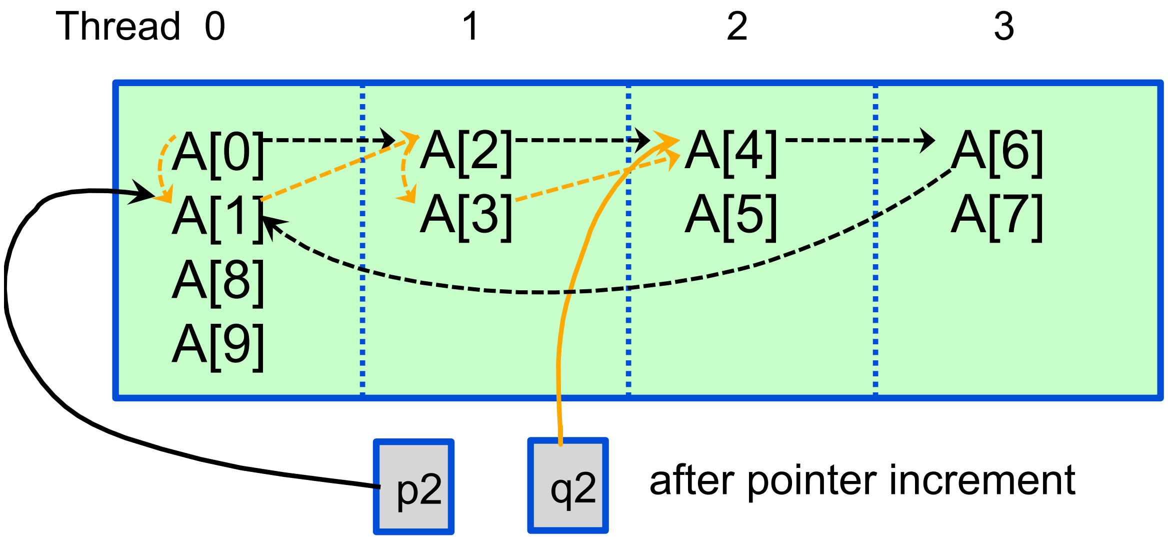
block size different from A

block size same as for A

```
if (MYTHREAD == 1) {
 p2 = (shared int *)&A[0];
 p2 += 4;
 q2 = &A[0];
 q2 += 4;
}
```

strange sequence

natural sequence



after pointer increment

- **Block size is a part of the variable‘s type**
- **One may cast between pointers with different block sizes**
  - pointer arithmetic follows blocking („phase“) of pointer (not pointee)!
  - cast changes the view but does not move any data
- **Consequences for libraries → see later**

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];
```

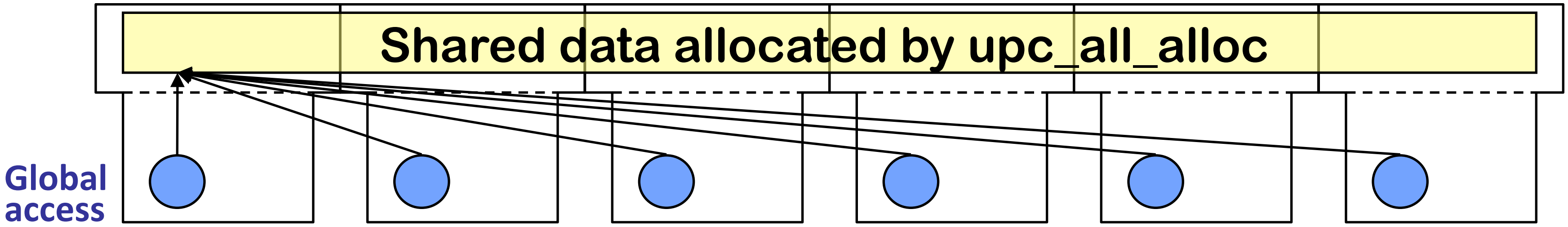
  - Example:

Compile-time constant!

Run-time expression!

```
shared [1] float *A;
A = (shared [1] float *) upc_all_alloc( n, sizeof(float) );
for (i=MYTHREAD; i<n; i+=THREADS) A[i] = …;
```

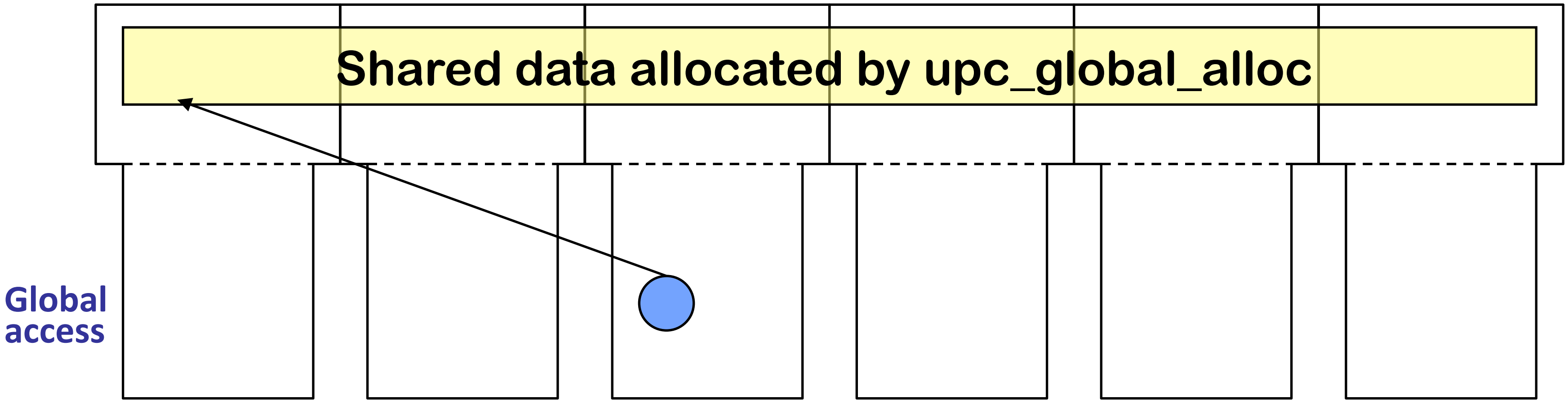
All threads may access A[i], i=0..n-1. Here, only the owning thread accesses A[i].



UPC dynamic Memory Allocation (2)

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

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



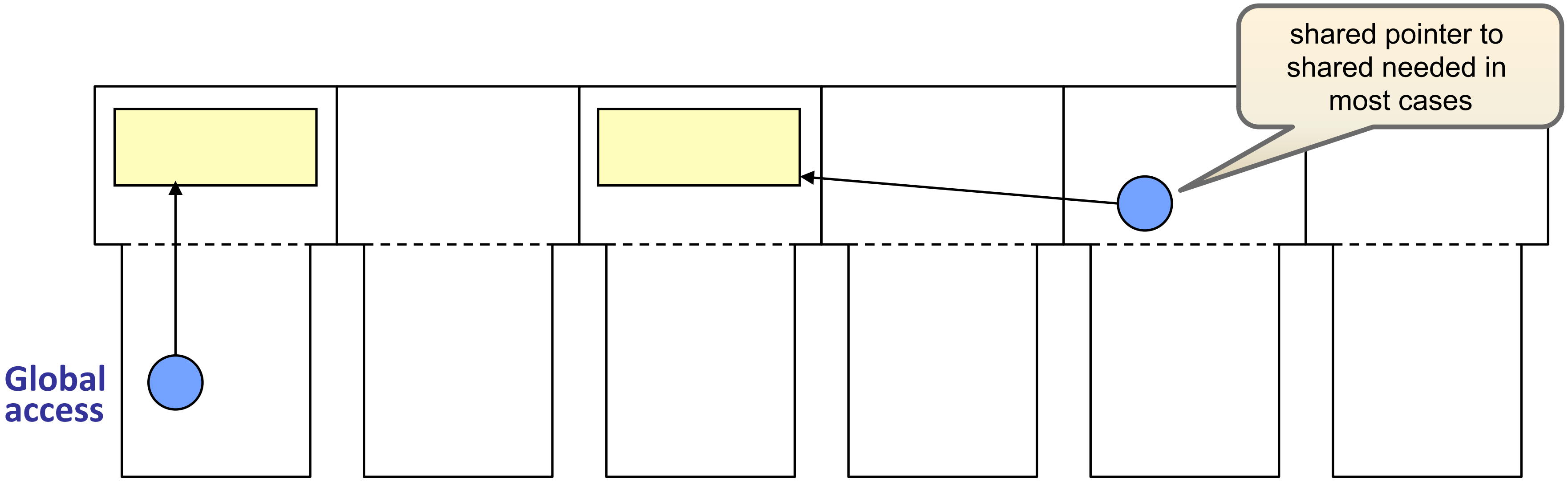
UPC dynamic Memory Allocation (3)

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



Common mistakes with dynamic allocation

- `shared int *p1 = upc_alloc(…);`
  - `p1` is cyclic, but the allocation is indefinite (all on calling thread)
  - Use of `p1[1]` might crash or might silently access wrong datum
  - Probably meant either of the following:

    ```
    shared int *p1 = upc_global_alloc(…); //cyclic
    shared [] int *p1 = upc_alloc(…); //indefinite
    ```
- `shared [2] int *p2 = upc_all_alloc(2, N*sizeof(int))`
  - Not *always* an error, but pretty often:
    first 2 is the ***size*** of a block, second 2 is the ***number*** of blocks
  - Probably meant either of the following:

    ```
    upc_all_alloc(N, 2*sizeof(int));   // 2*N elements
    upc_all_alloc(N/2, 2*sizeof(int)); // N elements
    ```
- Multiple calls to `upc_free()` for memory allocated by `upc_all_alloc()`
  - Even though all threads call `upc_all_alloc()`, only ***one*** object is allocated and it must be freed (at most) ***once.***
  - FUTURE: UPC 1.3 introduces `upc_all_free()` to help avoid this

skipped

UPC example with dynamic allocation

```
#include <upc.h>
#include <stdio.h>
#include <stdlib.h>
shared [] float * shared p4[THREADS];  // shared pointer array
                                       // to shared data
float *p1; // private pointer to private portion of shared data

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]);
      }
    }
  }
  return 0;
}
```

Each thread allocates a contiguous block of data

Each block may have a different length !!!

The addresses are stored in a p4-pointer, i.e., are accessible from all threads through p4

Local & *“efficient”* access through p1

After the barrier, all threads can access all locally stored data through p4. (Here an example with only thread 0 reading the data.)

skipped

UPC example with shared pointers

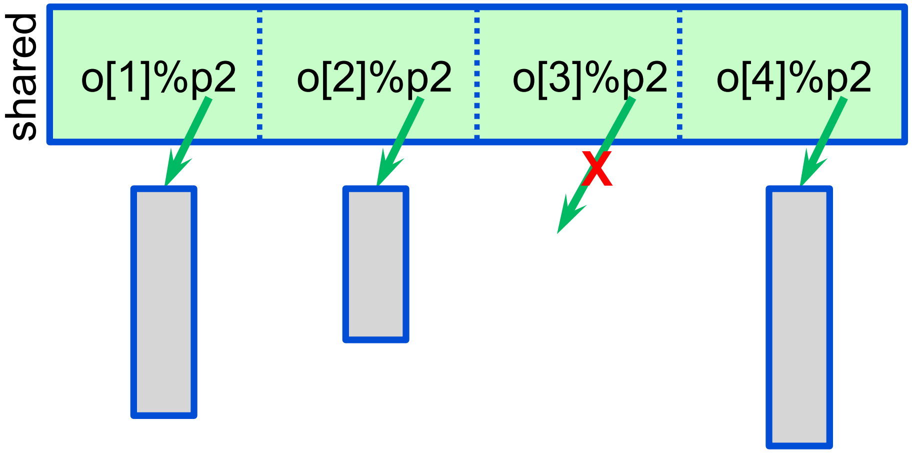
```
// same includes as on previous slide
shared [] float * shared p4[THREADS]; // shared pointer array
                                      // to shared data
float *p1; // private pointer to private portion of shared data
shared [] float *p2_neighbor; // private pointer to shared data

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++) {
    p1[i] = … /* local parts */
    p2_neighbor[i] = … /* neighbor data */
  }
  upc_barrier;

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

A p2-pointer can be used to access exactly one neighbor block

Integration of the type system
CAF dynamic components

- **Derived type component**
  - with **POINTER** attribute, or
  - with **ALLOCATABLE** attribute

  (don't care a lot about the differences for this discussion)



- **Definition/references**
  - **avoid** any scenario which requires **remote** allocation

- **Step-by-step:**
  1. **local** (non-synchronizing) allocation/association of component
  2. synchronize
  3. define / reference on remote image

remember earlier type definition

```
type(ctr) :: o[*]
:
if (this_image() == p) &
   allocate(o%p2(sz))
sync all
if (this_image() == q) &
   o[p]%p2 = <array of size sz>
end if
```

or `o%p2 => var`

`sz` same on each image?

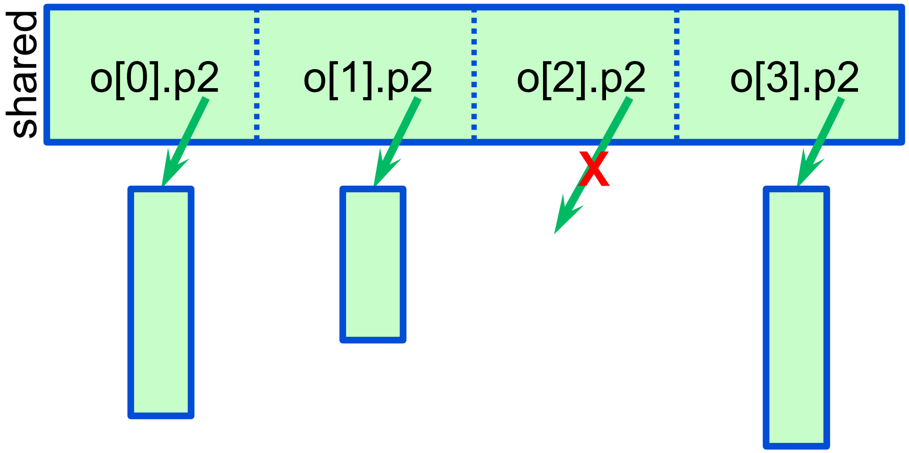
go to image p, look at descriptor, transfer (private) data

Integration of the type system
UPC pointer components

- **Type definition**

```
typedef struct {
  shared [] int *p2;
} Ctr;
```

dynamically allocated entity **should** be in shared memory area



- must avoid undefined results when transferring data between threads

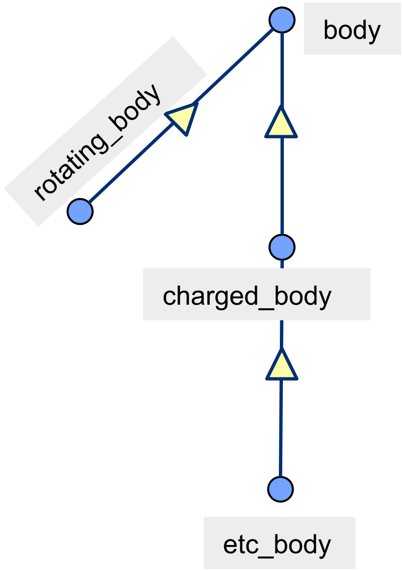
- **Similar step-by-step:**

```
shared [1] Ctr o[THREADS];

int main() {
  if (MYTHREAD == p) {
   o[MYTHREAD].p2 = (shared int *) \
           upc_alloc(SZ*sizeof(int));
  }
  upc_barrier;
  if (MYTHREAD == q) {
    for (i=0; i<SZ; i++) {
      o[p].p2[i] = … ;
    }
  }
}
```

- local (on thread p) allocation initializes pointer p2.
- program semantics the same as the CAF example on the previous slide

Fortran Object Model (1)

- **Type extension**



```
type :: body
  real :: mass
  : ! position,
velocity
end type

type, extends(body) ::
&
      charged_body
  real :: charge
end type

type(charged_bod
                pr



proton%mass = …
proton%charge = …
```

inherited

  – single inheritance (tree a DAG)

- **Polymorphic entities**
  – new kind of dynamic storage

declared type

```
class(body), &
     allocatable :: balloon

allocate(body :: balloon)
: ! send balloon on trip
if (hit_by_lightning()) then
  : ! save balloon data
  deallocate(balloon)
  allocate( &
      charged_body :: balloon)
  balloon = …
  ! balloon data + charge
end if
: ! continue trip if possible
```

typed allocation

must be an extension

  – change not only size, but also (dynamic) type of object during execution of program

Fortran Object Model (2)

- **Associate procedures with type**

```
type :: body
  : ! data components
  procedure(p), pointer :: print
contains
  procedure :: dp
end type

subroutine dp(this, kick)
  class(body), intent(inout) :: this
  real, intent(in) :: kick(3)
  : ! give body a kick
end subroutine
```

object-bound procedure (pointer)

type-bound procedure (TBP)

- – polymorphic dummy argument required for inheritance
- – TBP can be overridden by extension (must specify essentially same interface, down to keywords)

```
balloon%print => p_formatted
call balloon%print()
call balloon%dp(mykick)
```

`balloon` matches `this`

- **Run time type/class resolution**
  - – make components of dynamic type accessible

```
select type (balloon)
  type is (body)
    : ! balloon non-polymorphic here
  class is (rotating_body)
    : ! declared type lifted
  class default
    : ! implementation incomplete?
end select
```

polymorphic entity

- – at most one block is executed
- – use sparingly
- – same mechanism is used (internally) to resolve type-bound procedure calls

Object orientation and Parallelism (1)

- **Run time type resolution**

```
class(body), &
       allocatable :: asteroids[:]

allocate( rotating_body :: &
                    asteroids[*] )
! synchronizes
if (this_image == 1) then
  select type(asteroids)
    type is (rotating body)
    asteroids[2] = …
  end select
end if
```

required for coindexed access

  – allocation must guarantee **same** dynamic type on each image

- **Using procedures**

```
call asteroids%dp(kick)          ! Fine
call asteroids%print()           ! Fine
if (this_image() == 1) then
   select type(asteroids)
     type is (rotating_body)
     call asteroids[2]%print()   ! NO
     call asteroids[2]%dp(kick)  ! OK
   end select
end if
```

non-polymorphic

  – procedure pointers may point to a different target on each image
  – type-bound procedure is guaranteed to be the same

Object orientation and Parallelism (2)

- **Coarray type components**

```
type parallel_stuff
  real, allocatable :: a(:)[:]
  integer :: i
end type
```

must be `allocatable`

- **Usage:**

```
type(parallel_stuff) :: par_vec

allocate(par_vec%a(n)[*])
```

symmetric

  – entity must be:
    (1) non-allocatable, non-pointer
    (2) a scalar
    (3) not a coarray (because `par_vec%a` already is)

- **Type extension**
  – defining a coarray type component in an extension is allowed, but parent type also must have a coarray component

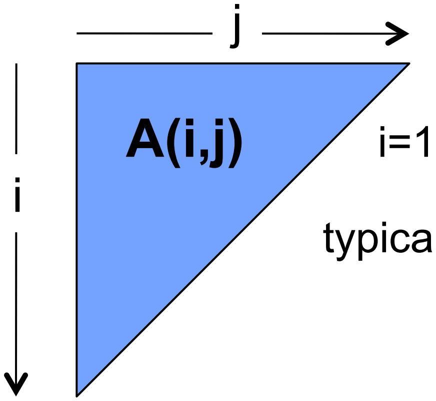
- **Restrictions on assignment**
  – intrinsic assignment to polymorphic coarrays (or coindexed entities) is prohibited

Major Differences between UPC and CAF

- **CAF**
  - declaration of shared entity requires additional codimension (“fragmented data view”).
  - Codimensions are very flexible (multi-dimensional).
- **UPC**
  - No codimensions (“global data view”).
  - PGAS-arrays are distributed and the array indices are mapped to threads.
  - Block-wise distribution hard to handle
    - Last index x[……][THREADS] implies round robin distribution
    - possibility of asymmetric distribution
  - Multiple variants of dynamic allocation

Second Exercise:
Handling a triangular matrix (1)

- **Consider a triangular matrix**



i=1..n, j=1..n-i+1

typically n >> number of tasks

  - suggested data structure

```
type :: tri_matrix
  real, allocatable :: row(:)
end type
```

Fortran

```
typedef struct {
  float *row;
  size_t row_len;
} Tri_matrix;
```

C

- **Procedure:**
  - make copy of **../triangular_matrix/triangular.f90** or **../triangular_matrix/triangular.c** to your working directory
  - the program reads in matrix size and a row index from the command line, it then sets up A(i,j) = i+j and prints out the specified row
  - parallelize this program in a manner that distributes data evenly across tasks
  - note that accesses to A can be kept purely local for this problem (which remote accesses will be needed?)

triangular

Handling a triangular matrix (2)

- **Example program run:**

#rows

row to print

```
aprun –n 3 ./triang.exe 23 20
 Row 20 on image 2: 21.0 22.0 23.0 24.0
Number of elements on image 2: 92
Number of elements on image 1: 100
Number of elements on image 3: 84
```

- **Suggestions:**
  - observe how location of row changes with number of image and row index
  - add the element count output as illustrated to the left

CAF: $a_{serial}(i) = a_{CAF}(\, i \,/\, nprocs\,)\, [mod(i, nprocs)]$ $i = 1,\ldots,n$

$a_{serial}(me + (i\_local-1)*nprocs) = a_{CAF}(i\_local)[me]$ $i\_local = 1,\ldots,rows\_per\_proc$

$me = 1,\ldots,nprocs$

UPC simple: $A_{serial}[i] = A_{UPC}[i]$

solutions/triangular_simple.upc

more general: $A_{serial}[i] = A_{UPC}[i\%THREADS\,]\, [i/THREADS]$ $i = 0,\ldots,n-1$

$A_{serial}[MYTHREAD + i\_local*THREADS] = A_{UPC}[MYTHREAD]\, [\, i\_local]$

$i\_local = 0,\ldots,rows\_per\_thread-1$

$MYTHREAD = 0,\ldots,THREADS-1$

solutions/triangular_cyclic.upc

- **Alternative exercise:**
  - each thread or image should print the specified row
  - for this alternative, start from the solution program
    - triangular_matrix/solutions/triangular.f90 (Fortran)
    - triangular_matrix/solutions/triangular.upc (UPC)

Solution program for alternative exercise:
triangular_printall.[upc|f90]

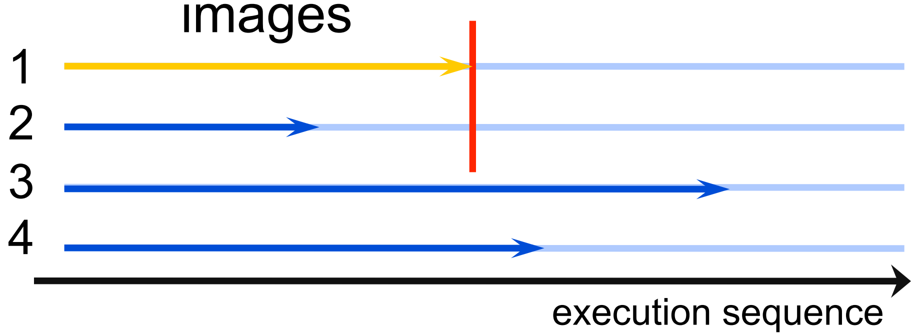
Advanced Synchronization Concepts

- Partial synchronization
  - mutual exclusion
  - split-phase barrier
- Collective operations
- Some parallel patterns and hints on library design:
  - parallelization concepts with and without halo cells
  - work sharing; distributed structures
  - procedure interfaces
- Hands-on session

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

Partial synchronization

- **Image subsets**
  - sometimes, it is sufficient to synchronize only a few images



  - CAF supports this:

```
if (this_image() < 3) then
  sync images ( (/ 1, 2 /) )
end if
```

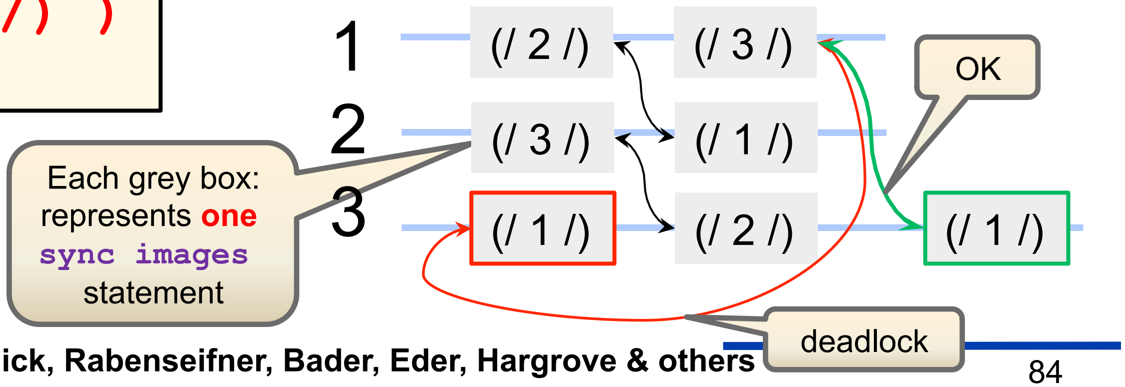
executing image implicitly included in image set

  - UPC does not explicitly support this; note that in

```
upc_barrier exp;
```

`exp` only serves as a label, with the same value on each thread

- **More than 2 images:**
  - need not have same image set on each image
  - but: eventually all image **pairs** must be resolved, else deadlock occurs



Each grey box: represents **one** `sync images` statement

Example: Simple Master-Worker

- **Scenario:**
  - one image sets up data for computations
  - others do computations

```
if (this_image() == 1) then
  : ! send data
  sync images ( * )
else
  sync images ( 1 )
  : ! use data
end if
```

„all images“

images 2 etc. don‘t mind stragglers

  - difference between `SYNC IMAGES (*)` and `SYNC ALL`: no need to execute from all images

- **Performance notes:**
  - sending of data by image 1

```
do i=2, num_images()
  a(:)[i] = …
end do
```

  - „push“ mode → a high quality implementation may implement non-blocking transfers
  - defer synchronization to image control statement

Partial synchronization: Best Practices

- **Localize complete set of synchronization statements**
  - **avoid** interleaved subroutine calls which do synchronization of their own

```
if (this_image() == 1) sync images (/ 2 /)
call mysub(…)
:
if (this_image() == 2) sync images (/ 1 /)
```

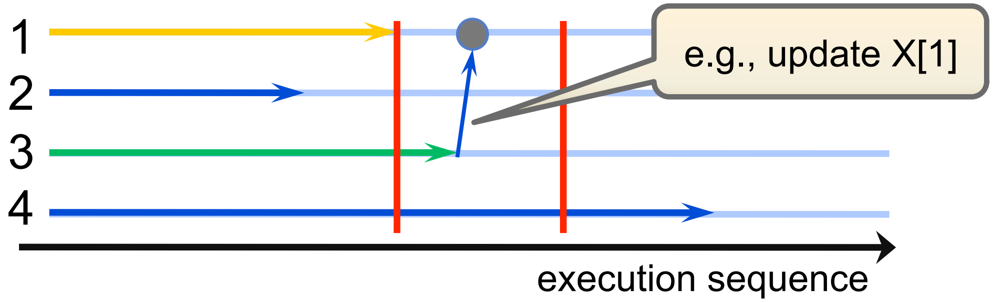
  - a very bad idea if subprogram does the following

```
subroutine mysub(…)
  :
  if (this_image() == 2) sync images (/ 1 /)
  :
end subroutine
```

  - may produce wrong results even if no deadlock occurs

Mutual Exclusion (simplest case)

- **Critical region**
  - In CAF only
  - block of code only executed by one image at a time



  - in arbitrary order

```
critical
   : ! statements in region
end critical
```

  - can have a name, but has no semantics associated with it

- **Subsequently executing images:**
  - segments corresponding to the code block ordered against one another
  - this does **not** apply to preceding or subsequent code blocks

  → may need additional synchronization to protect against race conditions

- **UPC:**
  - use locks (see following slides)

skipped

Memory fence

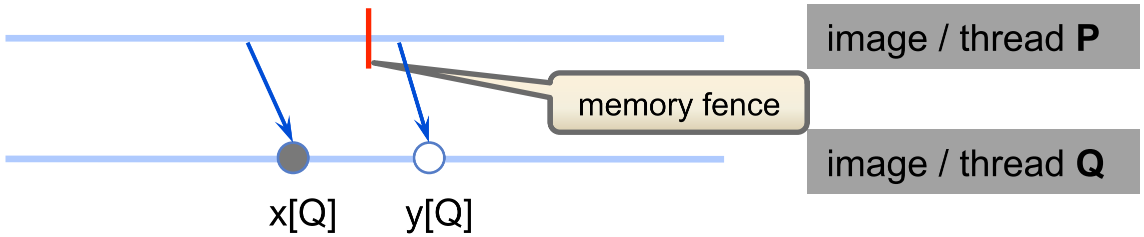
- **Goal: allow implementation of user-defined synchronization**
- **Prerequisite: subdivide a segment into two segments**
  - ensure memory operations are observed in-order

CAF:

```
sync memory
```

UPC: „null strict access“

```
upc_fence;
```



Note:
A memory fence is implied by **most other** synchronization statements

- **Assurance given by memory fence:**
  - operations on x[Q] and y[Q] via statements on P
  - action on x[Q] precedes action on y[Q] → code movement by compiler prohibited
  - P is subdivided into two segments / access epochs
  - **but:** segment on Q is unordered with respect to both segments on P

– skipped –

Atomic subroutines and atomic types

Remember synchronization rule for relaxed memory model:
A shared entity may not be modified and read from two different threads/images in unordered access epochs/segments
Atomic subroutines allow a **limited exception** to this rule

- **CAF:**

```
call ATOMIC_DEFINE(ATOM, VALUE)
call ATOMIC_REF(VALUE, ATOM)
```

  - **ATOM**: is a scalar coarray or co-indexed object of type **logical(atomic_logical_kind)** or **integer(atomic_int_kind)**
  - **VALUE**: is of same type as **ATOM**

- **Berkeley UPC extension:**

```
bupc_atomicI64_set_relaxed(ptr, value);
value = bupc_atomicI64_read_relaxed(ptr);
```

  - **shared int64_t *ptr;**
  - **int64_t value;**
  - unsigned and 32 bit integer types also available
  - „_relaxed“ indicates relaxed memory model
  - „_strict“ model also available

Semantics:

- ATOM/ptr always has a well-defined value if **only** the above subroutines are used
- for multiple updates (=definitions) on the same ATOM, **no assurance** is given about the order which is observed for references → programmers‘ responsibility

skipped

Example: Producer/Consumer

- **CAF:**

sync images ((/ p, q /)) would do the job as well

```
logical(ATOMIC_LOGICAL_KIND), save :: &
        ready[*] = .false.
logical :: val

me = THIS_IMAGE()
if (me == p) then
  : ! produce
  sync memory     ! A
  call ATOMIC_DEFINE(ready[q], .true.)
else if (me == q)
  val = .false.
  do while (.not. val)
    call ATOMIC_REF(val, ready)
  end do
  sync memory     ! B
  : ! consume
end if
```

segment P_i ends

segment Q_j starts

- **BUPC:**

roll-your-own partial synchronization

```
shared [] int32_t ready = 0;
int32_t val;

me = MYTHREAD;
if (me == p) {
  : // produce
  upc_fence;       ! A
  bupc_atomicI32_set_relaxed(&ready, 1);
} else if (me == q) {
  val = 0;
  while (! val) {
    val = \
    bupc_atomicI32_read_relaxed(&ready);
  }
  upc_fence;       ! B
  : // consume
}
```

  - memory fence: prevents reordering of statements (A), enforces memory loads (for coarrays, B)
  - atomic calls: ensure that B is executed after A

- **further atomic functions:**
  - swap, compare-and-swap, (fetch-and-) add, (fetch-and-)<*logical-operation*>
  - Will also be supported in future UPC 1.3 (with different syntax) and Coarray TS

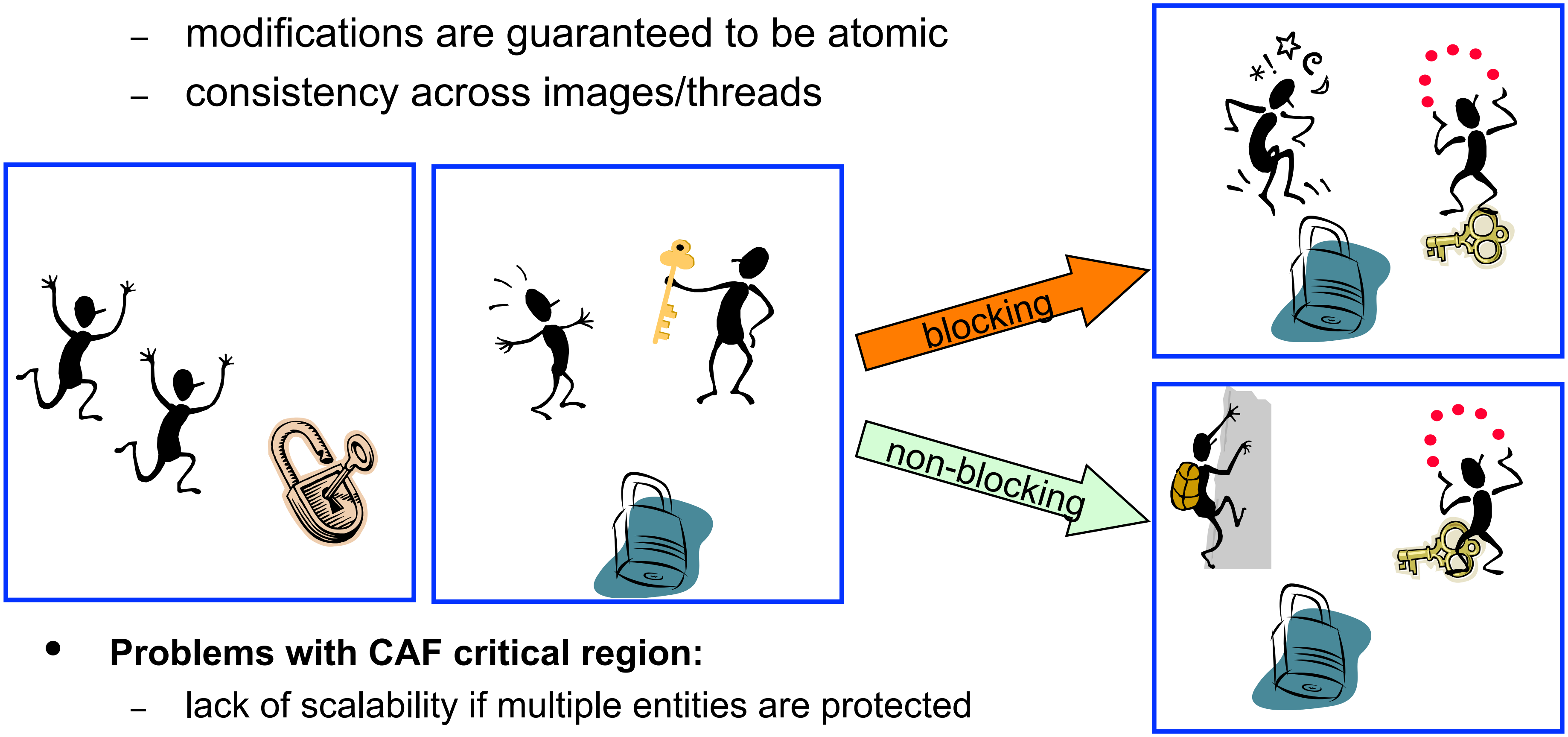
skipped

Recommendation

- **Functionality from the last three slides**
  - should be used only in exceptional situations
  - can be easily used in an unportable way (works on one system, fails on another) → beware

Locks – a more general mechanism for mutual exclusion

- **Coordinate access to shared (= sensitive) data**
  - sensitive data represented as “red balls”
- **Use a coarray/shared lock variable**
  - modifications are guaranteed to be atomic
  - consistency across images/threads



- **Problems with CAF critical region:**
  - lack of scalability if multiple entities are protected
  - updates to same entity in different parts of program

Usage of locks (1) – blocking

- **CAF:**
  - coarray lock variable

```
use, intrinsic :: iso_fortran_env

type(lock_type) :: lock[*]
! default initialized
! to unlocked

lock(lock[1])
: !  play with red balls
unlock(lock[1])
```

like `critical`, but more flexible

- as many locks as there are images, but typically only one is used
- lock/unlock: no memory fence, only **one-way** segment ordering

- **UPC:**
  - single pointer lock variable

```
#include <upc.h>

upc_lock_t *lock; // local pointer
                  // to shared entity

lock = upc_all_lock_alloc();

upc_lock(lock);
: // play with red balls
upc_unlock(lock);
upc_barrier; // prevent race vs. free
// single free from arbitrary thread
if (MYTHREADS == THREADS-1)
   upc_lock_free(lock);
```

collective call same result on each thread

- lock/unlock imply memory fence

Usage of locks (2) – nonblocking

- **CAF:**

```
use, intrinsic :: iso_fortran_env

type(lock_type) :: lock[*]
logical :: got_it
do
  lock(lock[2], &
        acquired_lock=got_it)
  if (got_it) exit
  : ! go climb that mountain
end do
: ! play with other red balls
unlock(lock[2])
```

- **UPC:**

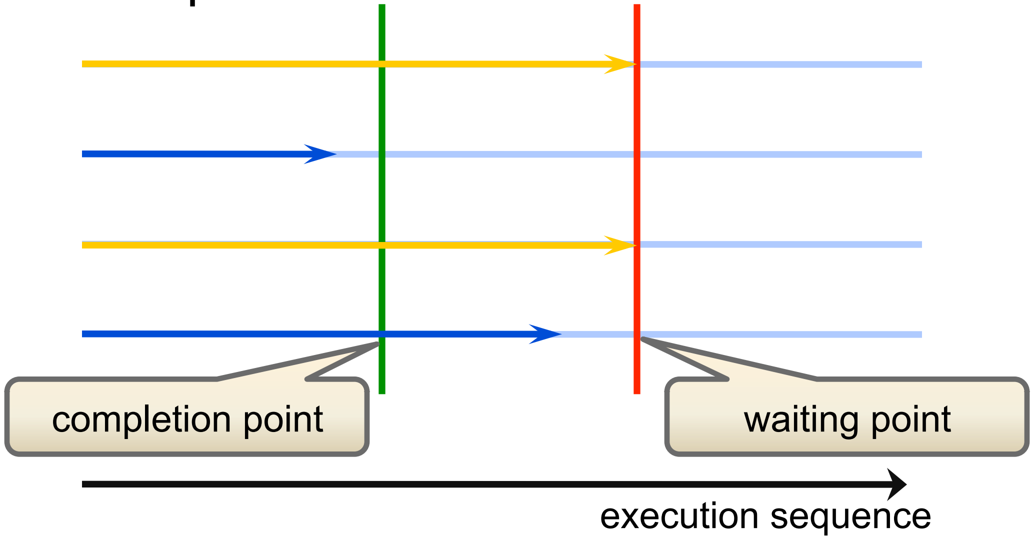
```
#include <upc.h>

upc_lock_t *lock; // local pointer
                  // to shared entity

lock = upc_all_lock_alloc();
for (;;) {
  if (upc_lock_attempt(lock)) break;
  : // go climb that mountain
}
: // play with red balls
upc_unlock(lock);
upc_barrier; // prevent race vs. free
// single free from arbitrary thread
if (MYTHREADS == THREADS-1)
   upc_lock_free(lock);
```

  - thread-individual lock generation is also possible (non-collective)
  - FUTURE: UPC 1.3 will include upc_all_lock_free() (with implicit barrier)

UPC: Split-phase barrier

- **Separate barrier completion point from waiting point**
  - this allows threads to continue computations after reaching the completion point → may reduce impact of load imbalance

```
for (…) a[n][i]= …;
upc_notify;
// do work (on b?) not
// involving a
upc_wait;
for (…) b[i]=b[i]+a[q][i];
```



  - completion of `upc_wait` once all threads reach `upc_notify`
  - collective – **all** threads must execute both calls in same order
- **CAF:**
  - presently does not have this facility in statement form
  - FUTURE: Notify/Query with events (non-collective though)

skipped

UPC: Memory consistency modes

- **How are shared entities accessed?**
  - relaxed mode → program **assumes** no concurrent accesses from different threads
  - strict mode → program **ensures** that accesses from different threads are separated, and **prevents** code movement across these synchronization points
  - relaxed is default; strict may have **large** performance **penalty**
- **Options for synchronization mode selection**
  - variable level:

    (at declaration

    or in a cast)

```
strict shared int flag = 0;
relaxed shared [*] int c[THREADS][3];
```

example for a spin lock

Thread q

```
c[q][i] = …;
flag = 1;
```

Thread p

```
while (!flag) {…};
… = c[q][j];
```

q has same value on thread p as on thread q

  - program level

```
{ // start of block
  #pragma upc strict
  … // block statements
}
// return to default mode
```

```
#include <upc_strict.h>
// or upc_relaxed.h
```

consistency mode on variable declaration **overrides** code section or program level specification

skipped

What strict memory consistency does and doesn‘t do for you

- **„strict“ cannot prevent all race conditions**
  - example: „ABA“ race

```
strict shared int flag;
int val, val1, val2;
```

thread 0

```
flag = 0;
upc_barrier;
flag = 1;
flag = 0;
```

thread 1

```
upc_barrier;
val = flag;
```

may end up with 0 or 1

- **„strict“ does not make a[i]+=j atomic (read/modify/write)**
- **„strict“ does assure that changes on (complex) objects appear in the same order on other threads**

thread 0

```
flag = 0;
upc_barrier;
flag = 1;
flag = 2;
```

thread 1

```
upc_barrier;
val1 = flag;
val2 = flag;
```

may obtain (val1 <= val2) but not (val1 > val2) e.g., (2, 1) or (2,0) are not possible

Advanced Synchronization Concepts

- o Partial synchronization
  - mutual exclusion
  - split-phase barrier
- o Collective operations
- o Some parallel patterns and hints on library design:
  - parallelization concepts with and without halo cells
  - work sharing; distributed structures
  - procedure interfaces
- o Hands-on session

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

Collective functions (1)

- **Two types:**
  - data redistribution (e.g., scatter, gather)
  - computation operations (reduce, prefix, sort)
- **Separate include file:**

```
#include <upc_collective.h>
```

- **Synchronization mode:**
  - constants of type `upc_flag_t`

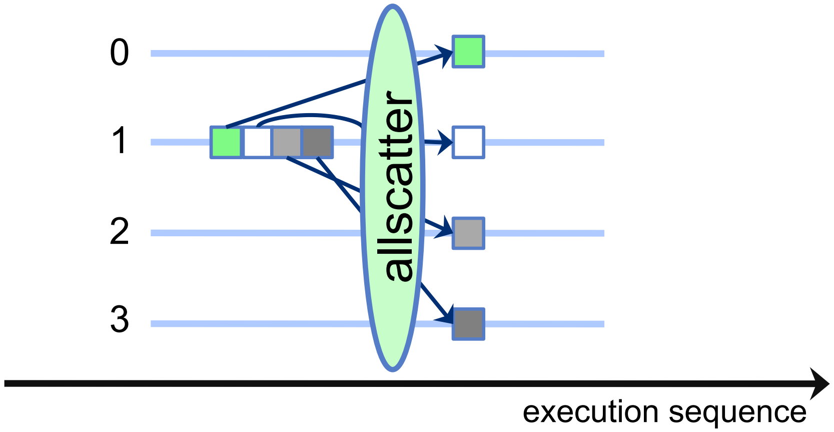
$$\texttt{UPC\_}\left\{\begin{matrix}\texttt{IN}\\\texttt{OUT}\end{matrix}\right\}\texttt{\_}\left\{\begin{matrix}\texttt{NOSYNC}\\\texttt{MYSYNC}\\\texttt{ALLSYNC}\end{matrix}\right.$$

- **IN/OUT:**
  - refers to whether the specified synchronization applies at the entry or exit to the call
- **Synchronization:**
  - NOSYNC – threads do not synchronize at entry or exit
  - MYSYNC – start processing of data only if owning threads have entered the call / exit function call only if all local read/writes complete
  - ALLSYNC – synchronize all threads at entry / exit to function
- **Combining modes:**
  - `UPC_IN_NOSYNC | UPC_OUT_MYSYNC`
  - `UPC_IN_NOSYNC` same as `UPC_IN_NOSYNC | UPC_OUT_ALLSYNC`
  - `0` same as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`

Collectives (2): Example for redistribution

- **UPC Allscatter**

```
void upc_all_scatter (
     shared void *dst,
     shared const void *src,
     size_t nbytes,
     upc_flag_t sync_mode);
```



  - **src** has affinity to a single thread
  - i-th block of size **nbytes** is copied to **src** with affinity to thread i

- **CAF:**
  - already supported by combined array and coarray syntax
  - „push“ variant:

```
if (this_image() == 2) then
  do i = 1, num_images
    b(1:sz)[i] = &
      a((i-1)*sz+1:i*sz)
  end do
end if
sync all
```

can be a non-coarray

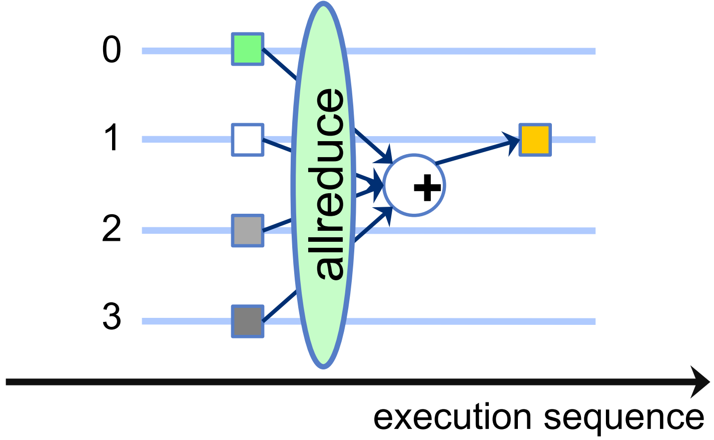
  - „pull“ variant:

```
me = this_image()
b(1:sz) = &
  a((me-1)*sz+1:me*sz)[2]
```

simpler, but no asynchronous execution possible

Collectives (3): Reductions

- **Reduction concept:**
  - distributed set of objects
  - operation defined on type



  - destination object resides in shared space
- **Availability:**
  - UPC only
  - CAF Future: TS will include some collectives

- **Reduction type codes**

C/UC – signed/unsigned char	**L/UL** – signed/unsigned long
S/US – signed/unsigned short	**F/D/LD** – float/double/long double
I/UI – signed/unsigned int	

- **Operations:**

Numeric	Logical	User-defined function
UPC_ADD	UPC_AND	UPC_FUNC
UPC_MULT	UPC_OR	UPC_NONCOMM_FUNC
UPC_MAX	UPC_XOR	
UPC_MIN	UPC_LOGAND	
	UPC_LOGOR	

  - are constants of type `upc_op_t`

Collectives (4): Reduction prototype

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

destination and source, respectively

number of elements of type T

source pointer block size,
or 0 for indefinite

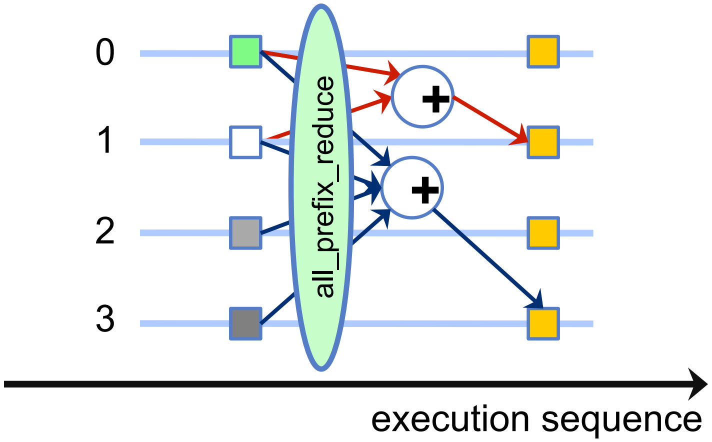
- `src` and `dst` may not be aliased
- replace `T` by type (C, UC, etc.)
- function argument will be `NULL` unless user-defined function is configured via `op`

Collectives (5): further functions

- **Redistribution functions**
  - upc_all_broadcast()
  - upc_all_gather_all()
  - upc_all_gather()
  - upc_all_exchange()
  - upc_all_permute()

→ **consult the UPC language specification for details**

- **Prefix reductions**
  - upc_all_prefix_reduce**T**()
  - semantics:



for UPC_ADD, thread i gets $\sum_{k=0}^{i} src[k]$

(thread-dependent result)

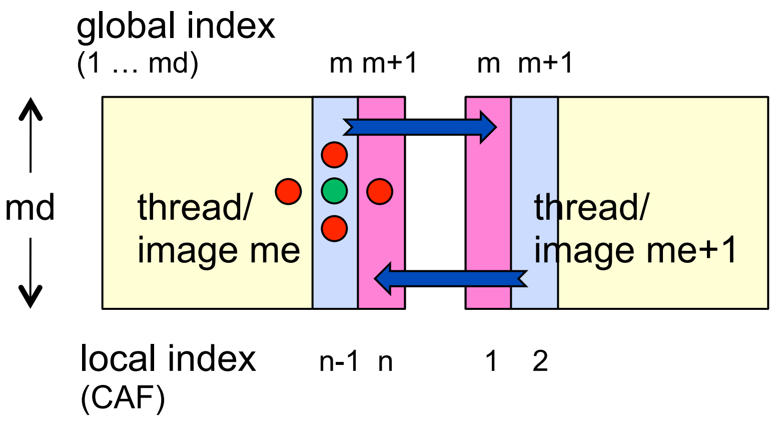
Advanced Synchronization Concepts

- Partial synchronization
  - mutual exclusion
  - split-phase barrier
- Collective operations
- Some parallel patterns and hints on library design:
  - parallelization concepts with and without halo cells
  - work sharing; distributed structures
  - procedure interfaces
- Hands-on session

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

Work sharing (3)
data exchange

- **Halo data** **(MPI like)**
  - context: stencil evaluation
  - example: Laplacian



(halo size is 1)

  - data exchange (blue arrows) required e.g. for iterative updates

- **CAF halo update**

```
real(dp),allocatable :: a_new(:,:)[*]
integer :: me, n, md
me = this_image()
: ! determine n, md
allocate(a_new(md, n)[*])
: ! initialize a
: ! calculate stencil a_new
sync all
if (me > 1) &
     a(:,1) = a_new(:,n-1)[me-1]
if (me < num_images()) &
     a(:,n) = a_new(:,2)[me+1]
sync all
: ! calculate next iteration
```

Assure stencil is done

Protect against subsequent write

  - uses „pull“ style („push“ also possible)
  - 1-d data distribution: not the most efficient way

Work sharing (4)

Avoiding the use of halo cells

- **Coarray Fortran**
  - interior region stencil is processed by local accesses
  - boundary region is treated separately, with remote accesses

```
: ! calculate interior a_new
sync all
! left neighbour image:
if (me > 1) a_new(2:n-1,1) = (&
  a(1:n-2,1) + a(3:n,1) + &
  a(2:n-1,2) + a(2:n-1,n)[me-1] &
  - 4 * a(2:n-1,1)) / (4.0_dk * dx)
! right neighbour image:
: ! (analogous procedure)
sync all
: ! copy a_new to a
: ! calculate next iteration
```

- **UPC**
  - can execute complete stencil update on shared array
  - easy to write, but may lose performance
  - cast to local pointer (for performance tuning) can only be done for interior region, then need to process boundary region separately with cross-thread accesses

- Easier to design (no halo data)
- But numerics replicated in communication part of code
- and compiler optimization and/or architecture support is required

→ see „Programming Styles with PGAS“

Subprogram interface

- **CAF coarray argument**

```
subroutine subr(n,w,x,y)
  integer :: n
  real :: w(n)[n,*] ! Explicit shape
  real :: x(n,*)[*] ! Assumed size
  real :: y(:,:)[*] ! Assumed shape
  :
end subroutine
```

- – corank specification is always assumed size
- – restrictions to **prevent** copy-in/out of coarray data:

  actual argument must be a coarray

  if dummy is not assumed-shape, actual must be contiguous

  VALUE attribute prohibited for dummy argument

- **UPC shared argument**

```
void subr(int n,
          shared float *w) {
  int i;
  float *wloc;
  wloc = (float *) &w[MYTHREAD];
  for (i=0; i<n; i++){
     … = wloc[i] + …
  }
  upc_barrier;
  // exchange data
  upc_barrier;
  // etc.
}
```

- – `subr` assumes local size is n
- – cast to local pointer for safety of use **and performance** if only local accesses are required
- – declarations with *fixed* block size > 1 also possible (default is 1, as usual)

Using the interface

- **CAF**

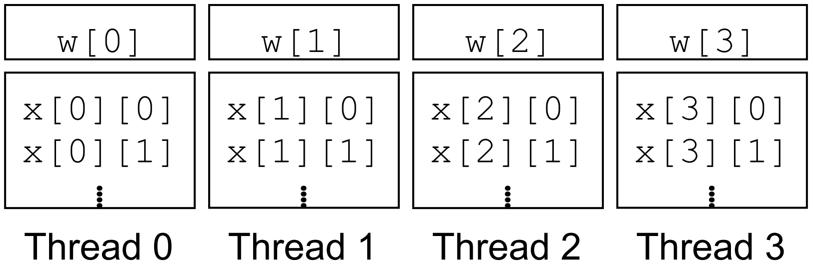
```
real :: a(ndim)[*], b(ndim,2)[*]
real, allocatable :: c(:,:,:)[:]
allocate(c(10,20,30)[*])

call subr(ndim, a, b, c(1,:,:))
```

  - a: corank mismatch is allowed (remapping inside subroutine)
  - c: assumed shape entity may be discontiguous

- **UPC**

```
shared [*] float x[THREADS][NDIM]
int main(void) {
  : // initialize x
  upc_barrier;
  subr(NDIM, (shared float *) x);
}
```

  - cast to cyclic to match the prototype
  - this approach of passing cyclic pointer and blocksize as arguments is a common solution to UPC library design.
  - cyclic is “good enough” in most cases because function can recover actual layout via pointer arithmetic
  - in this example w[i] aliases x[i][0]



Factory procedures

- **CAF:**
  allocatable dummy argument

```
subroutine factory(wk, …)
  real, allocatable :: wk(:)[:]
  : ! determine size n
  allocate(wk(n)[*])
  : ! fill wk with data
end subroutine
```

synchronizes all images

  - actual argument: must be allocatable, with matching type, rank **and corank**
  - procedure must be executed with all images

- **UPC:**
  shared pointer function result

```
shared *float factory(…) {
  shared float *wk;
  // determine size n to allocate
  wk = (shared float *)
    upc_all_alloc(THREADS,
                       sizeof(float)*n);
  : // fill wk with data
  return wk;
}
```

  - analogous functionality as for CAF is illustrated
  - remember: other allocation functions `upc_global_alloc` (single thread distributed entity), `upc_alloc` (single thread shared entity) do not synchronize

CAF: subprogram-local coarrays

- **Restrictions:**
  - no automatic coarrays
  - function result cannot be a coarray (avoid implicit SYNC ALL)
- **Consequence:**
  - require either the SAVE attribute

```
subroutine foo(a)
  real :: a(:)[*]
  real, SAVE :: wk_loc(ndim)[*]
  : ! work with wk_loc
end subroutine
```

storage preserved throughout execution

allow e.g., invocation by image subsets:

```
if (this_image() < num) then
  call foo(x)
else
  call bar(x)
end if
```

may have coindexed accesses to x in both foo and bar

  - or the ALLOCATABLE attribute:

```
recursive subroutine rec_process(a)
  real :: a(:)
  real, ALLOCATABLE :: wk_loc(:)[:]

  allocate(wk_loc(n)[*])
  :
  if (.not. done) &
     call rec_process(…)
end subroutine
```

requires execution by **all** images

allows recursive invocation, as shown in example (distinct entities are created)

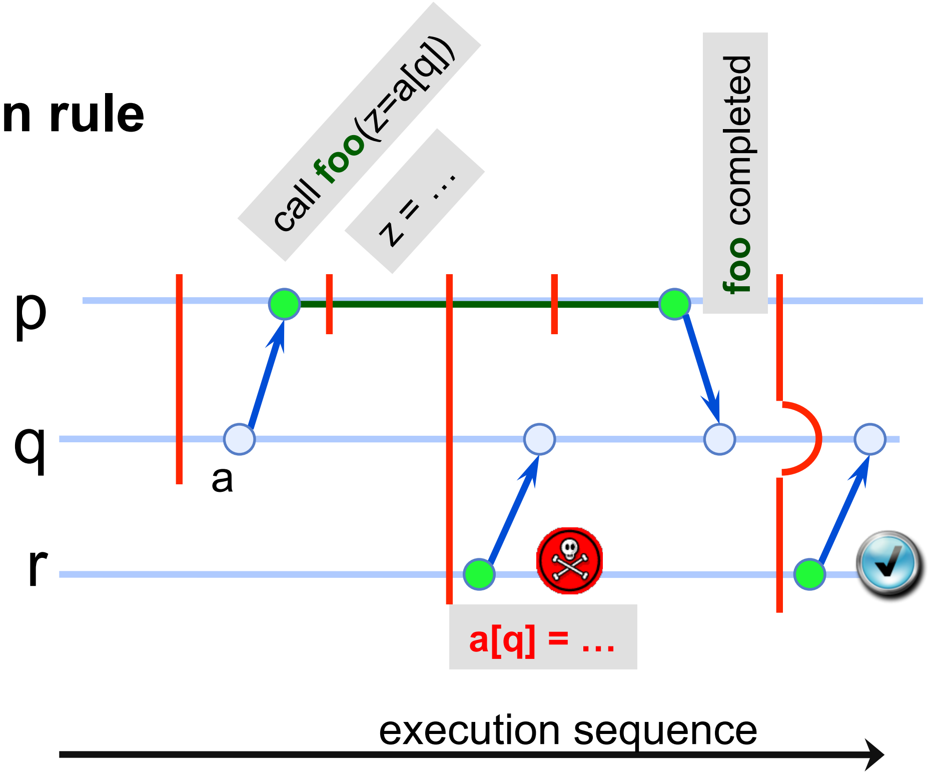
  - can also combine ALLOCATABLE with SAVE → a single entity, no automatic deallocation on return

CAF: Coindexed entities as actual arguments

- **Assumptions:**
  - dummy argument is not a coarray
  - it is modified inside the subprogram
  - therefore, typically copy-in/out will be required

→ **an additional synchronization rule is needed**

- **Note:**
  - UPC does not allow casting a remote shared entity to a private one

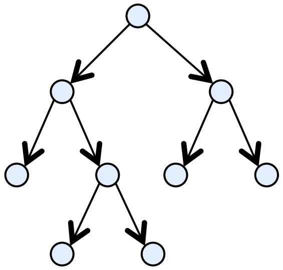


Distributed structures (1)

- **Irregular data structures**
  - example: binary tree
  - serial type definition:

```
typedef struct tree {
  struct tree *left;
  struct tree *right;
  Content *data;
};
typedef struct tree Tree;
```

  - each node contains:
    - data
    - information about siblings if present



- **Prerequisite**
  - ordering relation

```
int lessthan(Content *a, Content *b);
```

- **API:**
  - constructor and destructor
  - insertion routine

```
void insert(Tree *this, \
                  Content *stuff);
```

  - traversal (performs operations on all tree data)

```
void traverse(Tree *this, \
                 Params *op);
```

  - insertion and traversal work recursively

Distributed Structures (2)

- **Aim:**
  - concurrent processing of **distributed** binary tree
- **Type definition**

```
typedef struct tree {
  upc_lock_t *lk;
  shared struct tree *left;
  shared struct tree *right;
  shared Content *data;
};

typedef struct tree Tree;
```

use regular „serial“ type definition

  - add a lock component

```
int lessthan(shared Content *a,
                    Content *b);
```

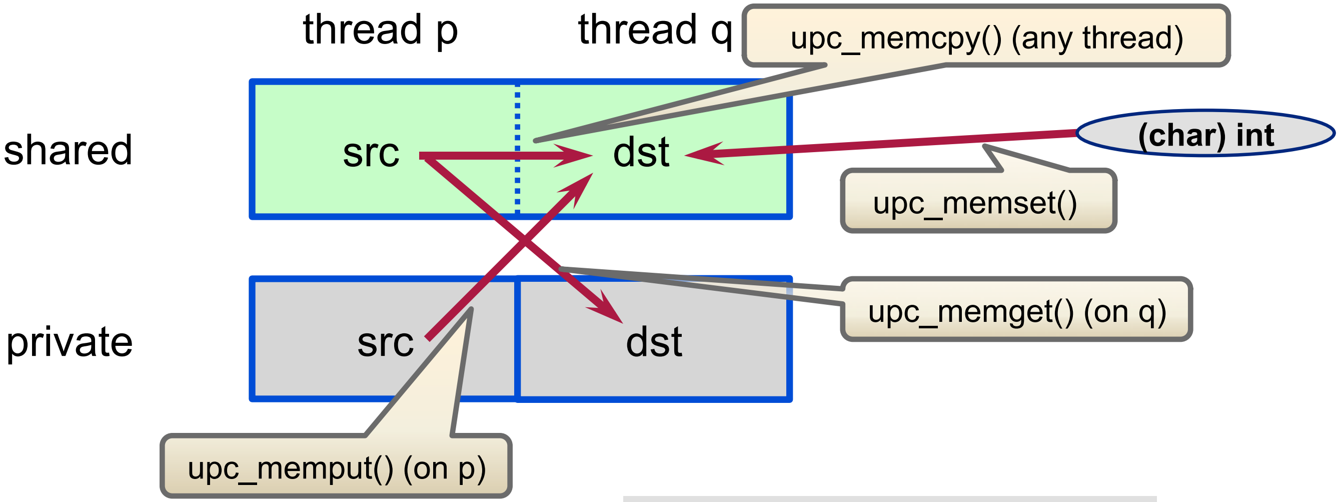
  - need to do remote copies for first argument

- **Constructor for Tree object**
  - to be called by **one** thread

```
shared Tree *Tree_init() {
  shared Tree *this;
  this = (shared Tree *)
          upc_alloc(sizeof(Tree));
  this->lk = upc_global_lock_alloc();
  this->data = (shared Content *)
          upc_alloc(sizeof(Content));
  this->left = this->right = NULL;
  return this;
}
```

  - initialize shared storage for lock and data components, NULL for children
  - `malloc()` of serial code is replaced by `upc_alloc()`

UPC: One-sided memory block transfer



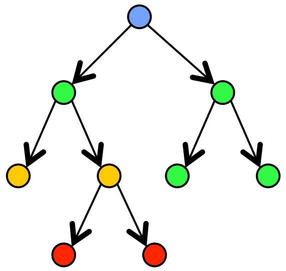
- **Available for efficiency**
  - operate in units of bytes
  - use restricted pointer arguments
- **Note:**
  - CAF array transfers should do this by default

prototypes from upc.h

```
void upc_memcpy(shared void *dst,
    shared const void *src, size_t n);
void upc_memget(void *dst,
    shared const void *src, size_t n);
void upc_memput(shared void *dst,
    void *src, size_t n);
void upc_memset(shared void *dst,
    int c, size_t n);
```

Distributed Structures (3)

- **Concurrent population**
  - locking ensures race-free processing



color ↔ thread number

```
void insert(shared Tree *this, Content *stuff) {
  upc_lock(this->lk);
  if ( this->left ) { // Interior node (contains data)
    upc_unlock(this->lk);
    if ( lessthan(this->data, stuff) ) {
      insert(this->left, stuff);
    } else {
      insert(this->right, stuff);
    }
  } else { // leaf node (no data value yet)
    this->left =  Tree_init();
    this->right = Tree_init();
    upc_memput(this->data, stuff, sizeof(Content));
    upc_unlock(this->lk);
  }
}
```

invoke constructor

copy object to (**remote**) shared entity

  - Invariant to simplify code (at the expense of storage): a node has EITHER
    - 2 children and „data“ field is used, OR
    - 0 children and „data“ points to allocated, but uninitialized, memory

Distributed Structures (4)

- **Assumption**
  - structure is written once or rarely (locking is expensive)
  - **many** operations performed on entries, in access epochs separated from `insert()` calls

```
void traverse(shared Tree *this,
              Params *op) {
if (this->data) { // non-empty node
   if (upc_threadof(this->data)
                    == MYTHREAD) {
     process((Content *)this->data, op);
   }
    traverse(this->left, op);
    traverse(this->right, op);
  }
}
```

guarantees locality

- to be complete, `traverse()` must be executed by all threads which called `insert()`, but not necessarily collectively
- **CAF:**
  - cannot easily implement this concept with coarrays
  - shared objects on one image only not supported
  - klugey workaround using pointer components of coarrays may be possible
- **Generalization**
  - implement e.g., tasking concept in UPC

Third exercise: Manual reduction and prefix reduction

- **This exercise is required for Fortran programmers**
  - UPC programmers could also make use of library function
- **Implement a global reduction facility for extended precision floating point numbers**
  - suggested interface:

```
real (dk) function caf_reduce(x, ufun)
  real(dk) intent(in) :: x
  interface
    real(dk) function ufun(a, b)
      real(dk), intent(in) :: a, b
    end function
  end interface
end function
```

not a coarray

user-provided function

- **Try the simplest implementation**
  - where do coarrays appear?
- **What do you need to change if you want to calculate a prefix reduction (`caf_prefix_reduce()`, same interface) instead?**

Reduction

Fourth Exercise: Heat conduction in 2 dimensions

- **Make a copy of serial programs into your working directory**
  - cp ../reduction_heat/heat_serial.c heat_upc.c
  - cp ../reduction/heat/heat_serial.f90 heat_caf.f90
- **Work items for parallelization:**
  - domain (data) decomposition (suggestion: use a 1-D decomposition for simplicity)
  - decide on shared data including halo, or local data with separate shared 1-D arrays for halo exchange (UPC only: use memory block transfer functions)
  - need a reduction operation to determine global convergence (use the code from the previous exercise)
  - halo exchange
  - organization of debug printout routine

Applications, Optimization, and Hybrid Programming

- Tools for Data Race Detection
- NAS parallel benchmarks
  - Optimization strategies in UPC
  - Hybrid concepts for optimization
- Hybrid programming
  - MPI allowances for hybrid models
  - Hybrid PGAS examples and performance/implementation comparison
- Hands-on session: optimization

UPC / Thrille
A Tool for Data Race Detection

- Observes shared memory accesses and synchronization behavior
- Can detect potential concurrency bugs in UPC programs
- Can actively control the schedule of threads to reproduce/fix bugs

- Run Thrille by adding -thrille=racer as a compiler option
- Potential races are reported in separate upct.race<num> files

- Compile with -trailler=tester, select race to reproduce using environent variable UPCT_RACE_ID=<num> and run

“Efficient Data Race Detection for Distributed Memory Parallel Programs,” SC11 Paper, Chang-Seo Park, Koushi Sen, Paul Hargrove, and Costin Iancu

The eight NAS parallel benchmarks (NPBs) have been written in various languages including hybrid for three

MG	Multigrid	Approximate the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method	
CG	Conjugate Gradient	Estimate smallest eigenvalue of sparse SPD matrix using the inverse iteration with the conjugate gradient method	
FT	Fast Fourier Transform	Solve a three-dimensional PDE using the fast Fourier transform (FFT)	
IS	Integer Sort	Sort small integers using the bucket sort algorithm	
EP	Embarrassingly Parallel	Generate independent Gaussian random variates using the Marsaglia polar method	
BT SP LU	Block Tridiagonal Scalar Pentadiag Lower/Upper	Solve a system of PDEs using 3 different algorithms	MZ

The NPBs in UPC are useful for studying various PGAS issues

- **Using customized communication to avoid hot-spots**
  - UPC Collectives do not support certain useful communication patterns
- **Blocking vs. Non-Blocking (Asynchronous) communication**
  - In FT and IS using non-blocking gave significantly worse performance
  - In MG using non-blocking gave small improvement
- **Benefits of message aggregation depends on the arch./interconnect**
  - In MG message aggregation is significantly better on Cray XT5 w/ SeaStar2 interconnect, but almost no difference is observable on Sun Constellation Cluster w/ InfiniBand
- **UPC – Shared Memory Programming studied in FT and IS**
  - Less communication but reduced memory utilization
- **Mapping BUPC language-level threads to Pthreads and/or Processes**
  - Mix of processes and pthreads often gives the best performance

Using customized communication to avoid hot-spots

- UPC Collectives might not support certain types of communication patterns (for example, vector reduction).
- Customized communication is sometimes necessary!
- Collective communication naïve approach (FT example):

```
for (i=0; i<THREADS; i++)
    upc_memget( … thread i … );
```

- Collective communication avoiding hot-spots:

```
for (i=0; i<THREADS; i++){
     peer = (MYTHREAD + i) % THREADS;
     upc_memget( … thread peer … );
 }
```

- Communication performance difference can exceed 50% (observed on Carver/NERSC – 2 quad-core Intel Nehalem cluster with Infiniband Interconnect)

Blocking vs. Non-Blocking (Asynchronous) communication

- **Berkeley UPC allows usage of non-blocking communication (for efficient computation/communication overlap):**
  - `upc_handle_t bupc_memget_async(void *dst, shared const void *src, size_t nbytes);`
    - starts communication
  - `void bupc_waitsync(upc_handle_t handle);`
    - wait for completion
  - Asynchronous versions of memcpy and memput also exist
- **Not always beneficial:**
  - Non-blocking communication can inject large number of messages into the network
  - Lower levels of the network stack (firmware, switches) can employ internal flow-control and reduce the bandwidth

Blocking vs. Non-Blocking (Asynchronous) communication (cont)

- FT – no communication/computation overlap possible, but non-blocking communication can be used:

```
bupc_handle_t handles[THREADS];
  for(i = 0; i < THREADS; i++) {
     peer = (MYTHREAD+i) % THREADS;
     handles[i] = bupc_memget_async( … thread peer … );
  }
  for(i=0; i < THREADS; i++)
       bupc_waitsync(handles[i]);
```

- Using non-blocking communication, FT (also IS) experiences up to 60% communication performance degradation. For MG we detected ~2% performance increase.
- Slowdown is caused by a large number of messages injected into the network (there is no computation that could overlap communication and reduce the injection rate)

In addition to asynchronous, one can study strided communication and message aggregation

- **Using strided communication is generally an improvement**
  - Again, BUPC has extensions for this purpose
- **Message aggregation reduces the number of messages, but introduces the packing/unpacking overhead**
- **Message aggregation increases programming effort**
- **Example:**

<u>Fine-grained communication</u>

Thread A → Thread B

```
for (i=0; i<n1; i++)
 upc_memput( &k[i],
   &u[i],
   n2 * sizeof( double ));
```

<u>Message Aggregation</u>

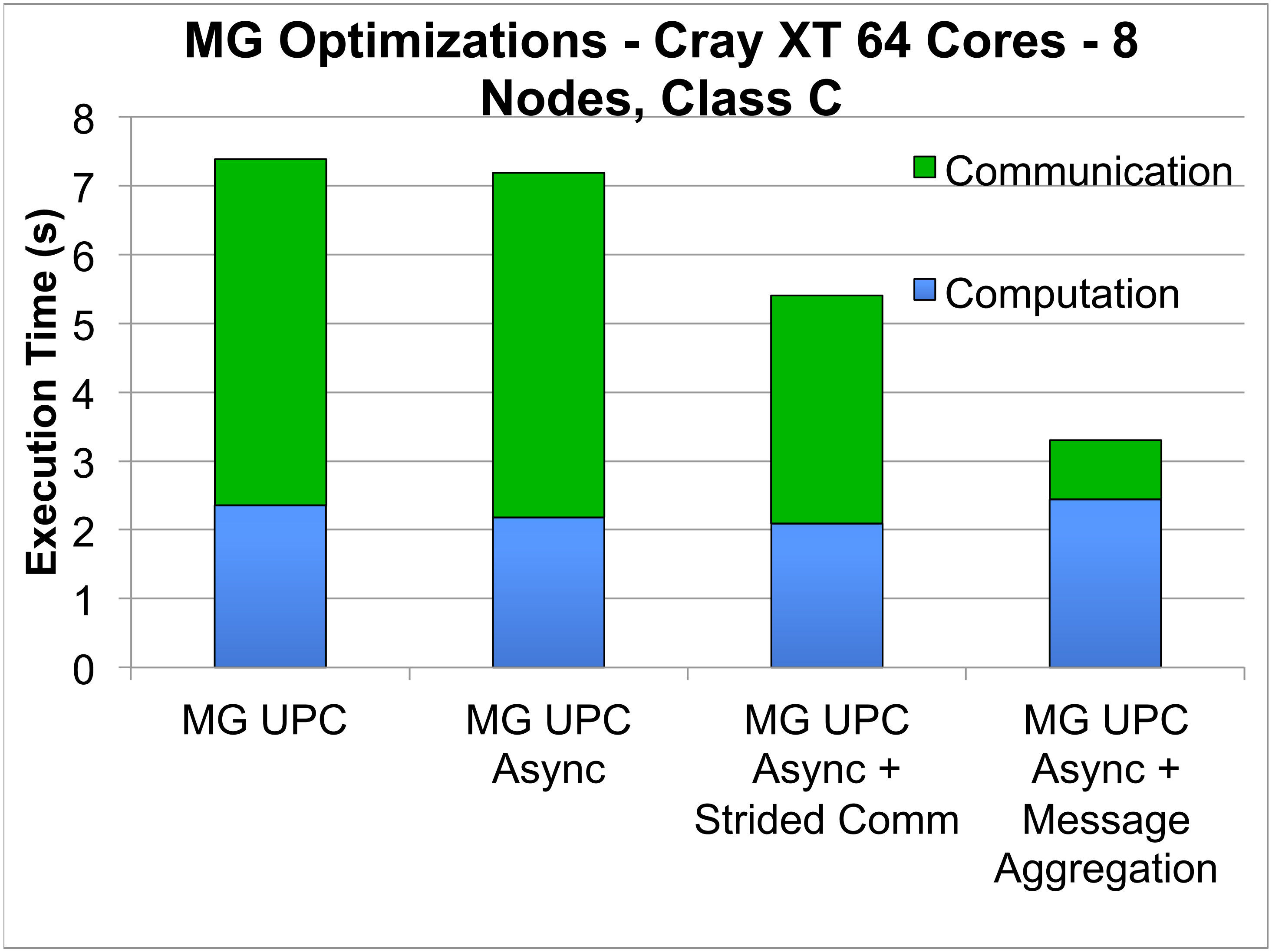
Thread A:

```
buff  = pack(u);
upc_memput( &k[0],
   &buff,
   n1*n2*sizeof(double));
upc_barrier;
```

Thread B:

```
upc_barrier;
unpack(k);
```

MG message aggregation is significantly better on Cray SeaStar2 interconnect



- MG message aggregation had almost no difference on Ranger InfiniBand interconnect

Class D NPBs have been run recently on two PF/s class machines at LRZ and LBL

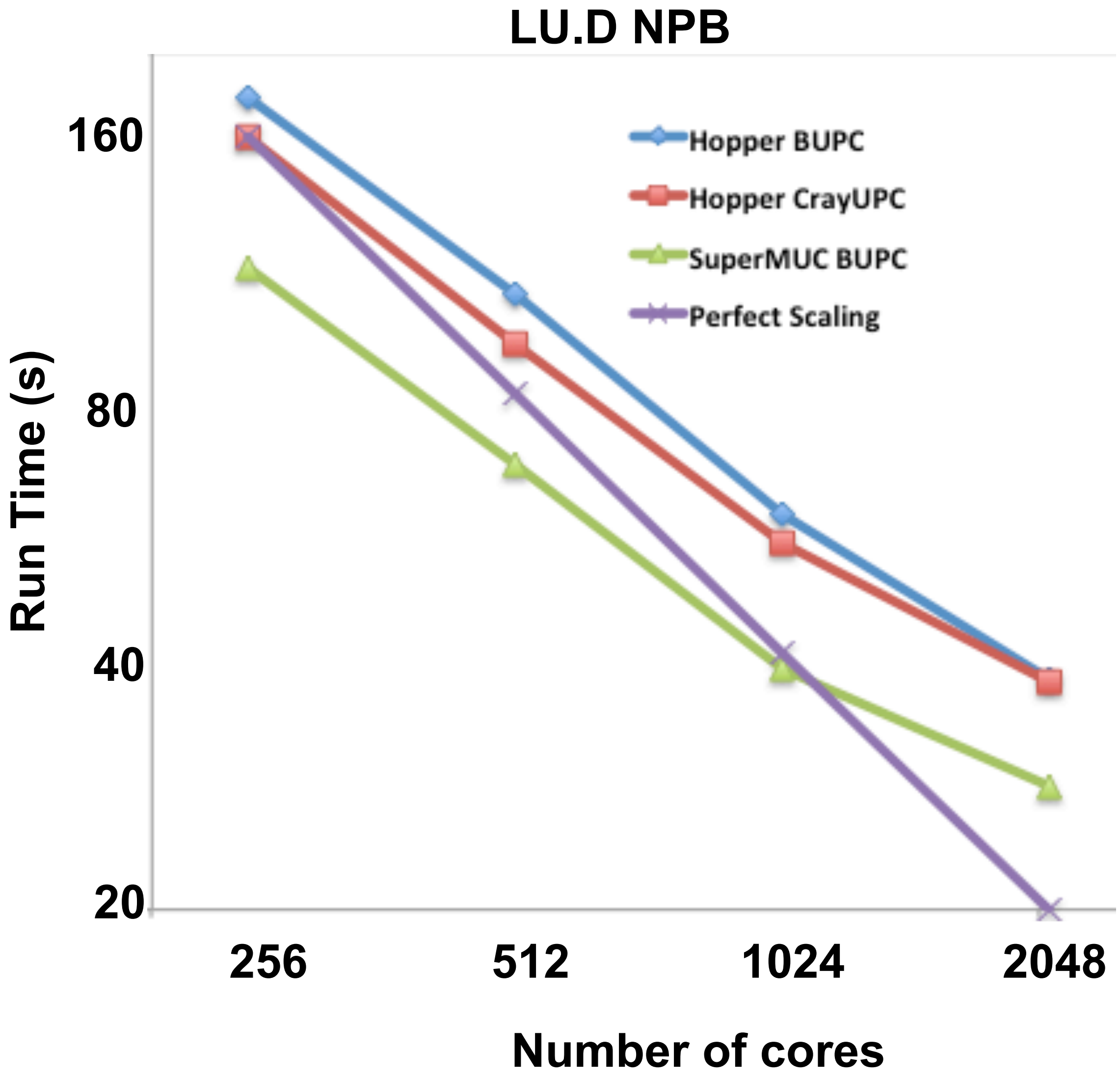
Property	SuperMuc	Hopper
Peak Performance	3.19 PF/s (#4)	1.28 PF/s (#16)
Number of Cores	147,456	153,216
Clock Speed	2.7 (3.5 Turbo) GHz	2.1 GHz
Interconnect	Infiniband FDR10	Gemini in 3D torus
Total Memory	288 TBytes	217 TBytes

MG.D 1024 cores Machine and Complier	Speed for 5 runs	No Flags	Message Aggregation	Message Agg + Strided Com
Hopper with Cray UPC	Median Gops/s	519.52	533.41 (+ 3%)	544.86 (+ 5%)
Hopper with Cray UPC	Avg Gops/s	519.22	527.98 (+ 2%)	546.19 (+ 5%)
Hopper with Cray UPC	SD Gops/s	3.55	12.93	6.61
SuperMUC with Berkeley UPC	Median Gops/s	879.8	1056.4 (+20%)	1026.5 (+ 17%)
SuperMUC with Berkeley UPC	Avg Gops/s	891.70	1034.5 (+16%)	1041.4 (+ 17%)
SuperMUC with Berkeley UPC	SD Gops/s	32.6	54.2	72.4

NPBs can used to study scalability as well as machine and complier effects

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization concepts

➢ Applications

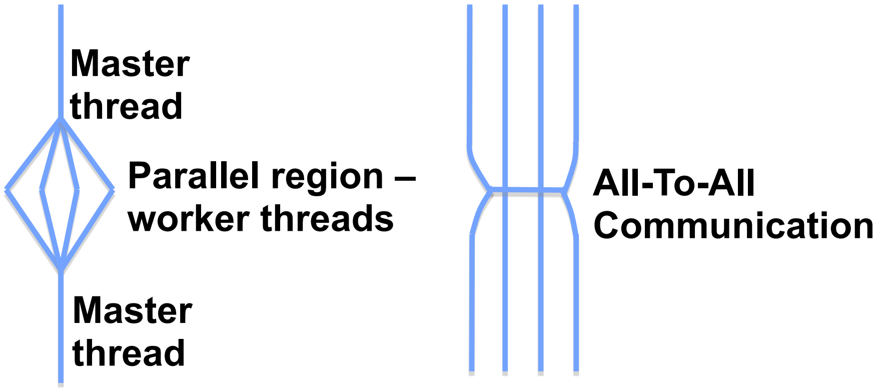


UPC – Hierarchical Shared Memory Programming reduces communication time

- **UPC designed for pure distributed or pure shared memory systems**
- **UPC capable of exploiting shared memory (OMP-like) programming style within a node (thus avoiding some explicit communication)**

OMP – Shared Memory style

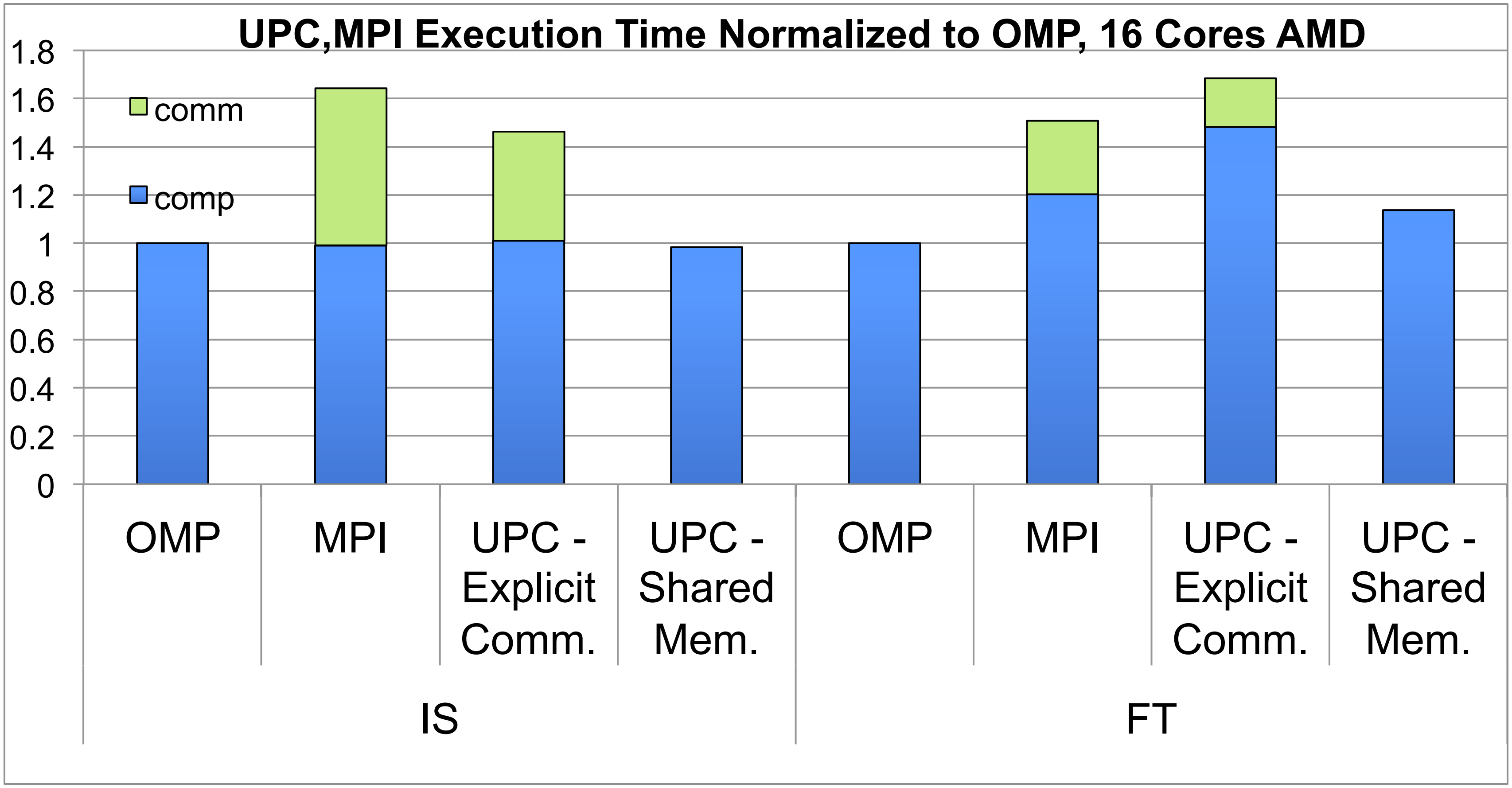
MPI – Explicit Communication



- **Drawback: reduced memory utilization (large fraction unusable)**
  - In the UPC hierarchical model, only the shared heap allocated by the master thread is used
  - In BUPC all threads have equally sized shared-heaps
  - In *any* UPC `upc_{all,global}_alloc()` allocate across all threads
  - Can result in large fraction of node memory potentially unusable
  - Careful data placement capable of increasing memory utilization
  - Berkeley is working on enabling uneven heap distribution in BUPC

Use of UPC shared memory reduced computation time by removing a transpose operation in FT

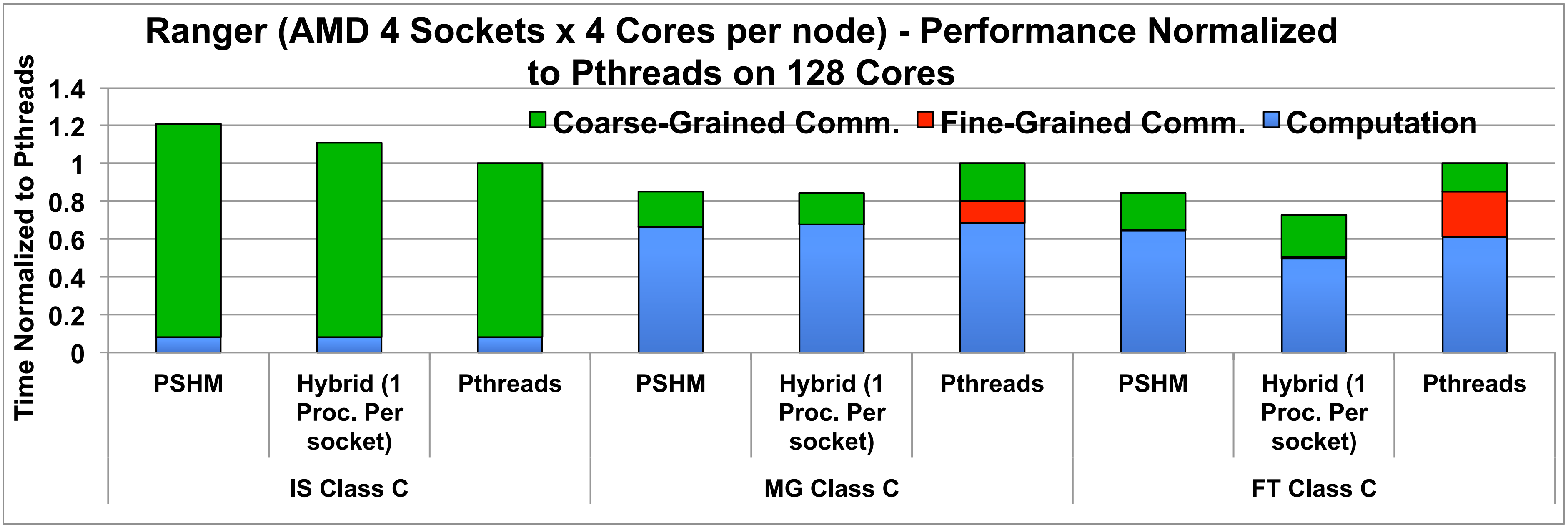
- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization concepts
- ➢ Applications



BUPC language-level threads can be mapped to Pthreads and/or Processes

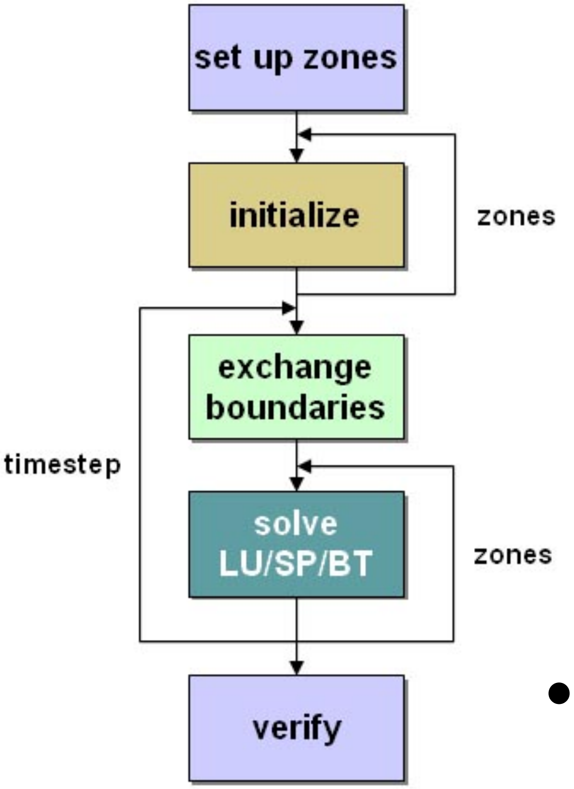
- **Pthreads – shared memory communication through shared address space**
- **Processes – shared memory communication through shared memory segments (POSIX, SysV or mmap(file)) called PSHM**
- **NPBs performance depends on Pthreads/Processes**
  - Pthreads share one network endpoint; PSHM has network endpoint per process
  - Due to sharing of one network endpoint, pthreads experience messaging contention, resulting in throttled injection rate
  - Processes (PSHM) can inject messages into the network faster (but large messages count may decrease effective bandwidth)
  - PSHM avoids contention overhead when interacting with external libraries/drivers
  - Contention and injection rate compete for dominance

Mix of processes and pthreads is often required for achieving the best performance



For FT the hybrid approach (1 process per socket and pthreads within a socket) is best and is a "reasonable" approach for the other NPBs

Some NAS Parallel Benchmarks have been written in multi-zone hybrid versions (currently with OpenMP)



	MPI/OpenMP Version
Time step	Sequential
Inter-zones	MPI Processes
Exchange boundaries	Call MPI
Intra zones	OpenMP

- Multi-zone versions of the NPSs LU,SP, and BT are available from:

www.nas.nasa.gov/Resources/Software/software.html

Figure adapted from Gabriele Jost, et al., ParCFD2009 Tutorial

Hybrid coding can yield improved performance for some benchmarks

- **BT-MZ: (Block-tridiagonal Solver)**
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance

Pure MPI: Load-balancing problems!

Good candidate for MPI+OpenMP

- **LU-MZ: (Lower-Upper Symmetric Gauss Seidel Solver)**
  - Size of the zones identical:
    - no load-balancing required
    - limited parallelism on outer level

Limited MPI Parallelism:
→ MPI+OpenMP increases Parallelism

- **SP-MZ: (Scalar-Pentadiagonal Solver)**
  - Size of zones identical
    - no load-balancing required

Load-balanced on MPI level: Pure MPI should perform best

Adapted from Gabriele Jost, et al., ParCFD2009 Tutorial

PGAS languages can also be combined with MPI for hybrid

- MPI is designed to allow *coexistence* with other parallel programming paradigms and uses the same *SPMD* model:
  - ➔ MPI and UPC or Coarrays can exist together in a program
- When mixing communications models, each will have its own progress mechanism and associated rules/assumptions
- Deadlocks can happen if some processes are executing blocking MPI operations while others are in “PGAS communication mode” and waiting for images (e.g., sync all)

  ➔ *"MPI phase" should end with MPI barrier, and a ”CAF phase" should end with a CAF barrier to avoid communication deadlocks*

There are differences between Rice and Cray CAF

- **CAF is becoming part of Fortran standard**
- **MPI *indexes* its processors from 0 to "number-of-processes – 1"**
  - Cray CAF indexes images from 1 to "num_images()".
  - Rice CAF indexes from 0 to "num_images() - 1")
- **Mixing OpenMP and CAF only works with Cray CAF**
  - Rice CAF interoperability still under development
  - OpenMP threads can execute CAF PUT/GET operations

We give one example of hybrid MPI and CAF interoperability

main.F90

```
program MPI_and_CAF

   integer ::   ntasks,ierr,rank,size
   integer,pointer,dimension(:) :: array

   call MPI_Init(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD,ntasks,ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)

   size = 1000
   allocate(array(1:size))
   array = 1

   call mpi_routine1(array)

   call MPI_BARRIER(MPI_COMM_WORLD,ierr)

   call caf_routine(rank,size,array)

   call MPI_BARRIER(MPI_COMM_WORLD,ierr)

   call mpi_routine2(array)

   deallocate(array)
   call MPI_FINALIZE(ierr)

end program MPI_and_CAF
```

mpi.F90

```
subroutine mpi_routine1…
subroutine mpi_routine2 …
```

caf.F90

```
subroutine caf_routine(mpi_rank,size,mpi_array)

   integer :: mpi_rank,size,world_rank,world_size
   integer,dimension(size ) :: mpi_array
   integer,allocatable :: co_array(:)[:]

   SYNC ALL ! Full barrier; wait for all images

   world_rank = THIS_IMAGE() ! equal to mpi_rank
   world_size = NUM_IMAGES()


   … ! some computation on mpi_array and co_array

   SYNC ALL

end subroutine caf_routine
```

```
# building for Hopper/Franklin @ NERSC:
module swap PrgEnv-pgi PrgEnv-cray
ftn –static –O3 –h caf caf.F90
ftn –static –O3 mpi.F90
ftn –static –O3 main.F90
ftn –static –o exec caf.o mpi.o main.o
```

Hybrid MPI and UPC is still under development on Cray platforms

- **Exercise is to download and compare three hybrid MPI-UPC versions of dot product**
  - Works on certain clusters but not yet on XT5 test platform

- **The three coding examples vary the level of nesting and number of instances of both models**
  - Flat model: provides a non-nested common MPI and UPC execution where each process is a part of both the MPI and the UPC execution
  - Nested-funneled model: provides an operational mode where only the master process per group gets an MPI rank and can make MPI calls
  - Nested-multiple model: provides a mode where every UPC process gets its own MPI rank and can make MPI calls independently.

Dot product coding from “Hybrid Parallel Programming with MPI and Unified Parallel C” by James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur

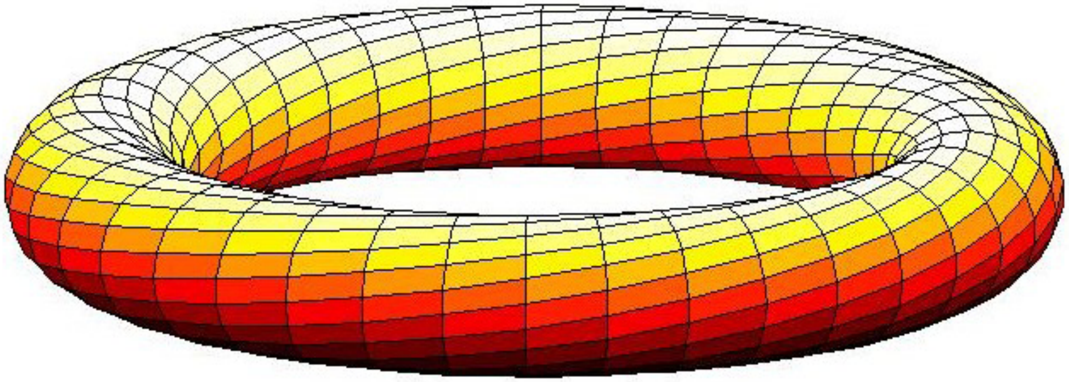
Exercise: Download, run, and time a hybrid MPI/CAF code example

- **Code is the communication intensive routine of a plasma simulation**

- **The simulation follows the trajectories of charged particles in a torus**

- **Due to the parallel domain decomposition of the torus, a huge number of particles have to be shifted at every iteration step from one domain to another using MPI**

- *Typically, 10% of each process' particles are sent to neighbor domain; 1% goes to "rank+2" and only a small fraction further.*



Compare differences in reduced code MPI and MPI-CAF benchmarks (coding/performance)

- MPI benchmark simulates the communication behavior of the code

- Iterates through an array of numbers in each domain with numbers that are a multiple of x (e.g. 10) being sent to “rank+1” and numbers which are a multiple of y (e.g. 100) being sent to “rank+2”

- The MPI-CAF benchmark follows exactly the algorithm but has been improved exploiting one-sided communication and image control techniques provided by CAF

The MPI version of the shifter benchmark

In order to precisely compare the performance of the MPI code vs. the CAF implementation, the MPI and CAF algorithm have to be in the same executable.

```
program MPI_CAF_ShifterBenchmark
  ……
  call mpi_benchmark(..)

  call MPI_BARRIER(MPI_COMM_WORLD,ierr)

  call caf_benchmark(..)

end program MPI_CAF_ShifterBenchmark
```

main.F90

caf_benchmark programming hints:

- use a multidimensional send-buffer (i.e., for each possible destination fill a send-vector)

- this send-vector has a fixed length := s

- if length of send-buffer(dest) == s then fire up a message to image “dest” and fill its receive queue

- for filling the 1D receive queue on a remote image use image control statements to ensure correctness (e.g. locks, critical sections, etc.)

```
subroutine mpi_benchmark()

 100: outer_loop = outer_loop  + 1
  do m=m0,array_size     ! use modulo operator on x and y for outer_loop==1
    if( is_shifted(array(m)) ) then  ! and just on y for outer_loop==2
      send_counter = send_counter + 1
      send_vector(send_counter) = m ! store position of sends
    endif

    MPI_Allreduce(send_counter,result) ! Stop when no numbers are sent
    if( result == 0 ) exit                           !  by all processors

    do i=1, send_counter  ! pack the send array
      send_array(i) = array( send_vector(i) )
    enddo

    fill_remaining_holes(array)

    MPI_Send_Recv(send_counter,recv_counter) ! send & recv new numbers
    MPI_Send_Recv(send_array, recv_array,..)

    do i=1, recv_counter  ! add the received numbers to local array
      array(a+i)=recv_array(i)
    enddo
    array_size = array_size - send_counter + recv_counter
    m0 = .. ! adapt array size, and the array starting position of next iteration
  enddo

end subroutine mpi_benchmark
```

caf.F90

Appendix

- Additional material on exercises
- Abstract
- Presenters
- Literature

https://fs.hlrs.de/projects/rabenseifner/publ/SC2012-PGAS.html

README – UPC on Cray XE…: UPC / PGI

```
Initialization:  module load bupc

Interactive PBS shell:
In the SC tutorial
  qsub -I -q special -lmppwidth=24,mppnppn=24, \
                     walltime=00:30:00 -V

Again to the working directory:
  cd $PBS_O_WORKDIR

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

Parallel Execution:
  upcrun -n 1 -cpus-per-node 24 ./myprog
  upcrun -n 2 -cpus-per-node 24 ./myprog
  upcrun -n 4 -cpus-per-node 24 ./myprog
```

Exercise 1

Exercise 2

Exercise 3/4

README – UPC on Cray XE…: Cray UPC

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

Interactive PBS shell:
In the SC tutorial
  qsub -I -q special -lmppwidth=24,mppnppn=24, \
                    walltime=00:30:00 -V

Again to the working directory:
  cd $PBS_O_WORKDIR

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

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

Exercise 1

Exercise 2

Exercise 3/4

README – UPC on Cray XE…: Cray Fortran

Initialization:

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

Interactive PBS shell:

In the SC tutorial

```
  qsub -I -q special -lmppwidth=24,mppnppn=24, \
                     walltime=00:30:00 -V
```

Again to the working directory:

```
  cd $PBS_O_WORKDIR
```

Compilation:

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

Parallel Execution:

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

Exercise 1

Exercise 2

Exercise 3/4

hello_upc_1.c and hello_caf_1.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()
numprocs = NUM_IMAGES()
if (myrank == 1) print *, 'hello world'
write (*,*) 'I am image number',myrank, &
          & ' of ',numprocs,' images'
end program hello
```

Exercise 1

Dynamic entities:
triangular.f90

- Matrix object declaration and initialization code

```fortran
type(tri_matrix), allocatable :: a(:)[:]
:
me = this_image() ; nproc = num_images()
rows_per_proc = n / nproc
if (mod(n, nproc) > 0) &
       rows_per_proc = rows_per_proc + 1
allocate(a(rows_per_proc)[*])
! initialize matrix A(i, j) = i + j
i_local = 1
n_elem = 0
do i = me, n, nproc
  allocate(a(i_local)%row(n - i + 1))
  do j = 1, n - i + 1
     a(i_local)%row(j) = real(i) + real(j)
  end do
  n_elem = n_elem + n - i + 1
  i_local = i_local + 1
end do
```

- Solution programs available as
  - ../triangular_matrix/ solutions/triangular.f90 (Fortran)
  - ../triangular_matrix/ solutions/triangular.upc (UPC)

Exercise 2

Manual reduction: mod_reduction_simple.f90

- **Singleton coarray g as module variable**

```
real(dk) function  &
         caf_reduce(x, ufun)
  real(dk), intent(in) :: x
  procedure(rf) :: ufun

  if (this_image() == 1) then
     g = x
     sync images(*)
  else
     sync images(1)
     critical
        g[1] = ufun(x,g[1])
     end critical
  end if
  sync all
  caf_reduce = g[1]
  sync all ! protect against
     ! subsequent write of g
  end function caf_reduce
```

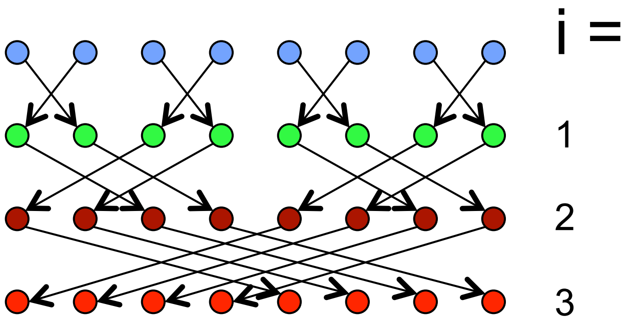
- **Prefix reduction**
  - pipelined execution („John Reid‘s ladder“)

```
real(dk) function &
         caf_prefix_reduce(x, ufun)
    real(dk), intent(in) :: x
    procedure(rf) :: ufun
    integer :: me
    me = this_image()
    if (me == 1) then
       g = x
       caf_prefix_reduce = x
    else
       sync images ((/me,me-1/))
       g = ufun(x,g[me-1])
       caf_prefix_reduce = g
    end if
    if (me < num_images()) &
       sync images ((/me,me+1/))
    sync all ! protect against
       ! subsequent write of g on 1
end function caf_prefix_reduce
```

Exercise 3

Manual reduction (2)

- **Programs from previous slide**
  - are not the most efficient solutions
  - alternative: „butterfly pattern“
- **Power-of-two version**
  - illustrative code based on tutorial material by Bob Numrich



```
real(dk) :: g[*]
! global variable
```

- **Files for study:**
  - reduction_heat/solutions/mod_reduction*

```
real(dk) function caf_reduce(x, ufun)
  real(dk), intent(in) :: x
  procedure(rf) :: ufun
  real(kind=8) :: work
  integer :: n,bit,i,mypal,dim,me
  : ! dim is log2(num_images())
  : ! dim == 0 trivial
  g = x
  bit = 1; me = this_image(g,1) - 1
  do i=1, dim
     mypal = xor(me,bit)
     bit = shiftl(bit,1)
     sync all
     work = g[mypal+1]
     sync all
     g = ufun(g,work)
  end do
  caf_reduce = g
  sync all ! against subsequent write on g
end function
```

Appendix: Abstract

PGAS (Partitioned Global Address Space) languages offer both an alternative to traditional parallelization approaches (MPI and OpenMP), and the possibility of being combined with MPI for a multicore Applications model. In this tutorial we cover PGAS concepts and two commonly used PGAS languages, **Coarray Fortran (CAF, as specified in the Fortran standard)** and the extension to the C standard, **Unified Parallel C (UPC)**.

Exercises exercises to illustrate important concepts are interspersed with the lectures. Attendees will be paired in groups of two to accommodate attendees without laptops. Basic PGAS features, syntax for data distribution, intrinsic functions and synchronization primitives are discussed.

Additional topics include parallel programming patterns, future extensions of both CAF and UPC, and hybrid programming. In the hybrid programming section we show how to combine PGAS languages with MPI, and contrast this approach to combining OpenMP with MPI. Real applications using hybrid models are given.

Presenters

- **Dr. Alice Koniges** is a Physicist and Computer Scientist at the National Energy Research Scientific Computing Center (NERSC) at the Berkeley Lab. Previous to working at the Berkeley Lab, she held various positions at the Lawrence Livermore National Laboratory, including management of the Lab's institutional computing. She recently led the effort to develop a new code that is used predict the impacts of target shrapnel and debris on the operation of the National Ignition Facility (NIF), the world's most powerful laser. Her current research interests include parallel computing and benchmarking, arbitrary Lagrange Eulerian methods for time-dependent PDE's, and applications in plasma physics and material science. She was the first woman to receive a PhD in Applied and Computational Mathematics at Princeton University and also has MSE and MA degrees from Princeton and a BA in Applied Mechanics from the University of California, San Diego. She is editor and lead author of the book "Industrial Strength Parallel Computing," (Morgan Kaufmann Publishers 2000) and has published more than 80 refereed technical papers.

Presenters

- **Dr. Katherine Yelick** is the Director of the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory and a Professor of Electrical Engineering and Computer Sciences at the University of California at Berkeley. She is the author or co-author of two books and more than 100 refereed technical papers on parallel languages, compilers, algorithms, libraries, architecture, and storage. She co-invented the UPC and Titanium languages and demonstrated their applicability across architectures through the use of novel runtime and compilation methods. She also co-developed techniques for self-tuning numerical libraries, including the first self-tuned library for sparse matrix kernels which automatically adapt the code to properties of the matrix structure and machine. Her work includes performance analysis and modeling as well as optimization techniques for memory hierarchies, multicore processors, communication libraries, and processor accelerators. She has worked with interdisciplinary teams on application scaling, and her own applications work includes parallelization of a model for blood flow in the heart. She earned her Ph.D. in Electrical Engineering and Computer Science from MIT and has been a professor of Electrical Engineering and Computer Sciences at UC Berkeley since 1991 with a joint research appointment at Berkeley Lab since 1996. She has received multiple research and teaching awards and is a member of the California Council on Science and Technology and a member of the National Academies committee on Sustaining Growth in Computing Performance.

Presenters

- **Dr. Rolf Rabenseifner** studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without losses to full MPI functionality. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum and since December 2007 he is in the steering committee of the MPI-3 Forum. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of Parallel Computing - Training and Application Services at HLRS. He is involved in MPI profiling and benchmarking e.g., in the HPC Challenge Benchmark Suite. In recent projects, he studied parallel I/O, parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. In workshops and summer schools, he teaches parallel programming models in many universities and labs in Germany.
  - Homepage: http://www.hlrs.de/people/rabenseifner/
  - List of publications: https://fs.hlrs.de//projects/rabenseifner/publ/
  - International teaching: https://fs.hlrs.de//projects/rabenseifner/publ/#tutorials

Presenters

- **Dr. Reinhold Bader** studied physics and mathematics at the Ludwigs-Maximilians University in Munich, completing his studies with a PhD in theoretical solid state physics in 1998. Since the beginning of 1999, he has worked at Leibniz Supercomputing Centre (LRZ) as a member of the scientific staff, being involved in HPC user support, procurements of new systems, benchmarking of prototypes in the context of the PRACE project, courses for parallel programming, and configuration management for the HPC systems deployed at LRZ. As a member of the German delegation to WG5, the international Fortran Standards Committee, he also takes part in the discussions on further development of the Fortran language. He has published a number of contributions to ACMs Fortran Forum and is responsible for development and maintenance of the Fortran interface to the GNU Scientific Library.

  Sample of national teaching:

  - LRZ Munich / RRZE Erlangen 2001-2011 (5 days) - G. Hager, R. Bader et al: Parallel Programming and Optimization on High Performance Systems
  - LRZ Munich (2009-2011) (5 days) - R. Bader: Advanced Fortran topics - object-oriented programming, design patterns, coarrays and C interoperability
  - LRZ Munich (2010) (1 day) - A. Block and R. Bader: PGAS programming with coarray Fortran and UPC

Presenters

- **Dr. David Eder** is a computational physicist and group leader at the Lawrence Livermore National Laboratory in California. He has extensive experience with application codes for the study of multiphysics problems. His latest endeavors include ALE (Arbitrary Lagrange Eulerian) on unstructured and block-structured grids for simulations that span many orders of magnitude. He was awarded a research prize in 2000 for use of advanced codes to design the National Ignition Facility 192 beam laser currently under construction. He has a PhD in Astrophysics from Princeton University and a BS in Mathematics and Physics from the Univ. of Colorado. He has published approximately 80 research papers.

Literature

- **UPC references**
  - UPC Language specification, by the UPC Consortium: http://upc.gwu.edu/docs/upc_specs_1.2.pdf
  - UPC Manual, by Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, Tarek El-Ghazawi, May 2005 http://upc.gwu.edu/downloads/Manual-1.2.pdf
  - UPC Distributed Memory Programming, by Tarek El-Ghazawi, Bill Carlson, Thomas Sterling, and Katherine Yelick, Wiley & Sons, June 2005
- **Coarray references**
  - Coarrays in the next Fortran Standard, by John Reid
WG5 paper N1824, April 21, 2010, ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
  - Fortran 2008 draft international standard
  - Coarray compendium, by Andy Vaught, http://www.g95.org/compendium.pdf