

---

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

Alice E. 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 and Robert Preissl – Lawrence Berkeley National Laboratory



Tutorial S10 at SC10,  
November 14, 2010, New Orleans, LA, 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 and Hybrid Programming** (Alice Koniges, David Eder)
  - Exercise 5 (hybrid)
- **Appendix**

<https://fs.hlrs.de/projects/rabenseifner/publ/SC2010-PGAS.html>  
(the pdf includes additional “skipped” slides)

# Basic PGAS Concepts

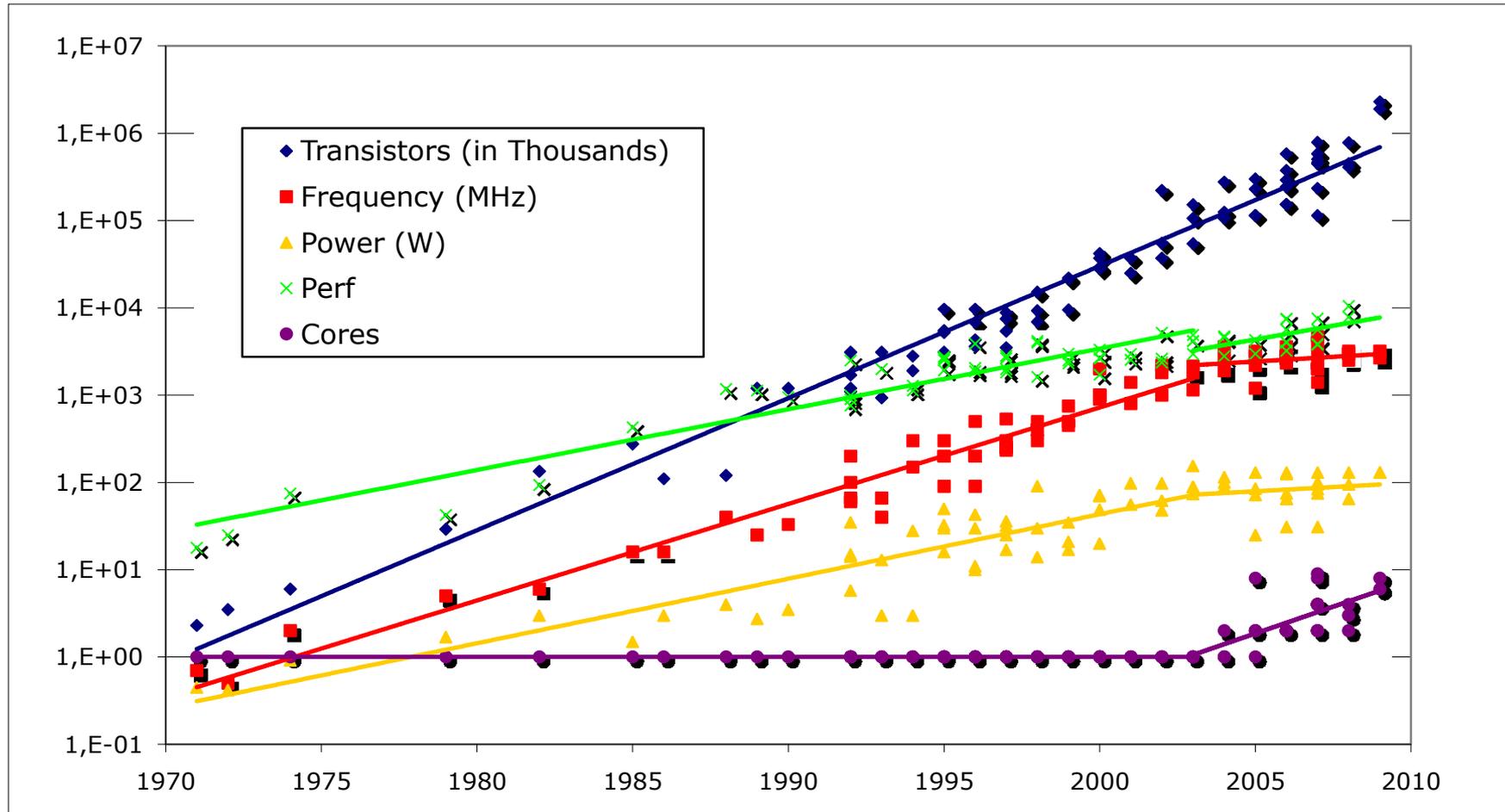
- Trends in hardware
- Execution model
- Memory model
- Run time environments
- Comparison with other paradigms
- Standardization efforts

Hands-on session: First UPC and CAF exercise

<https://fs.hlr.de/projects/rabenseifner/publ/SC2010-PGAS.html>

# Moore's Law with Core Doubling Rather than Clock Speed

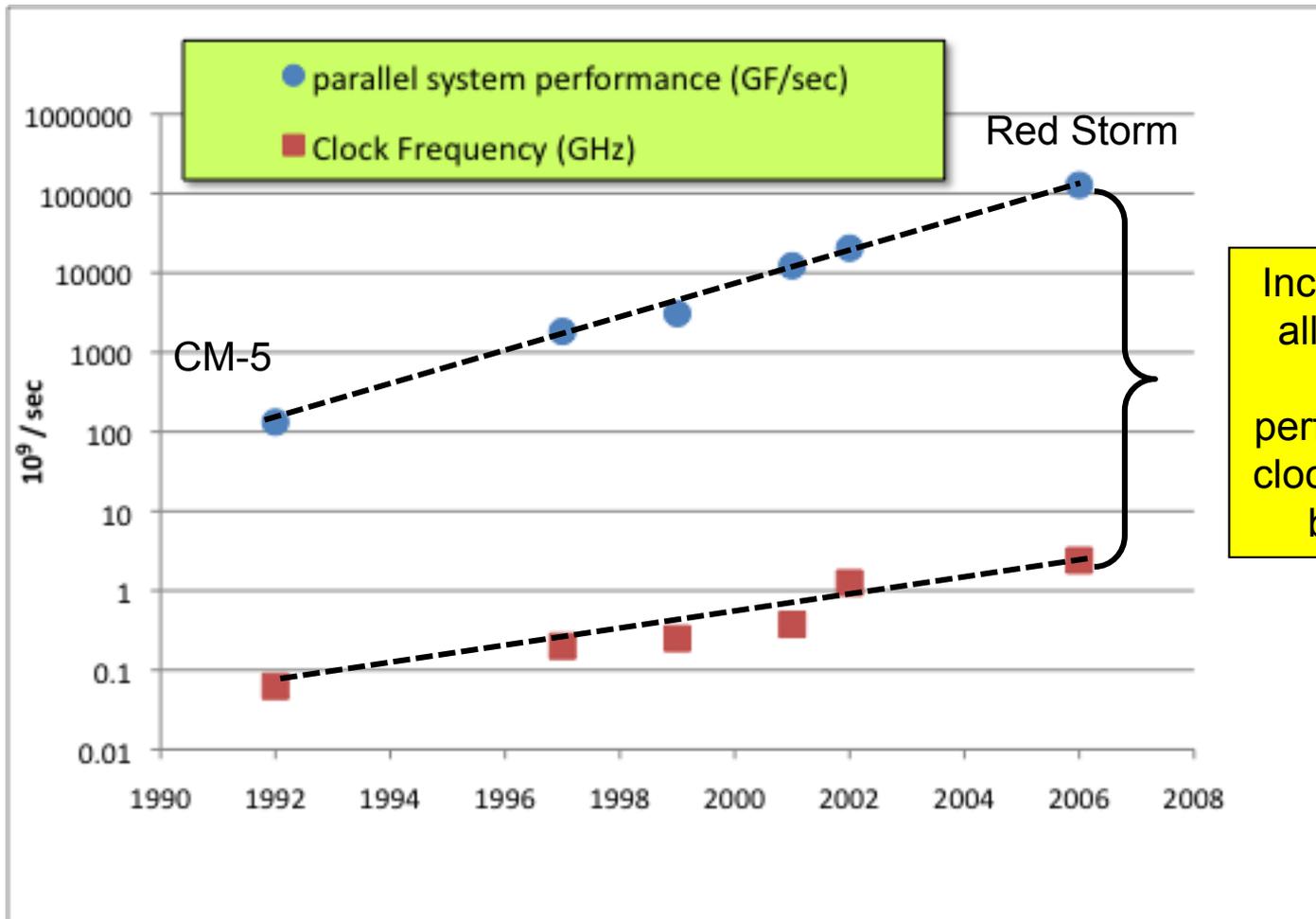
- Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming and outlook



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

# Concurrency was Part of the Performance Increase in the Past

- Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming and outlook



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

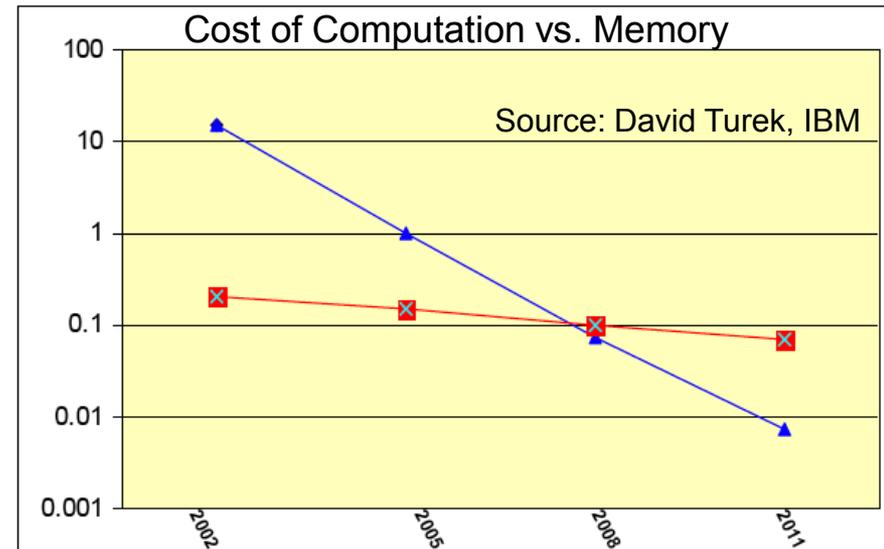
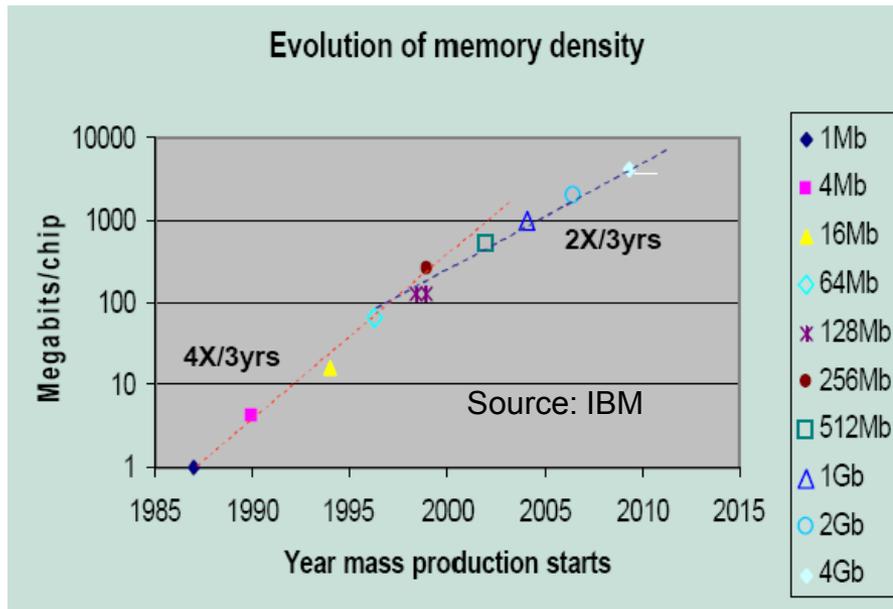
*and power, resiliency, programming models, memory bandwidth, I/O, ...*

# Memory is Not Keeping Pace

- Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming and outlook

## 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

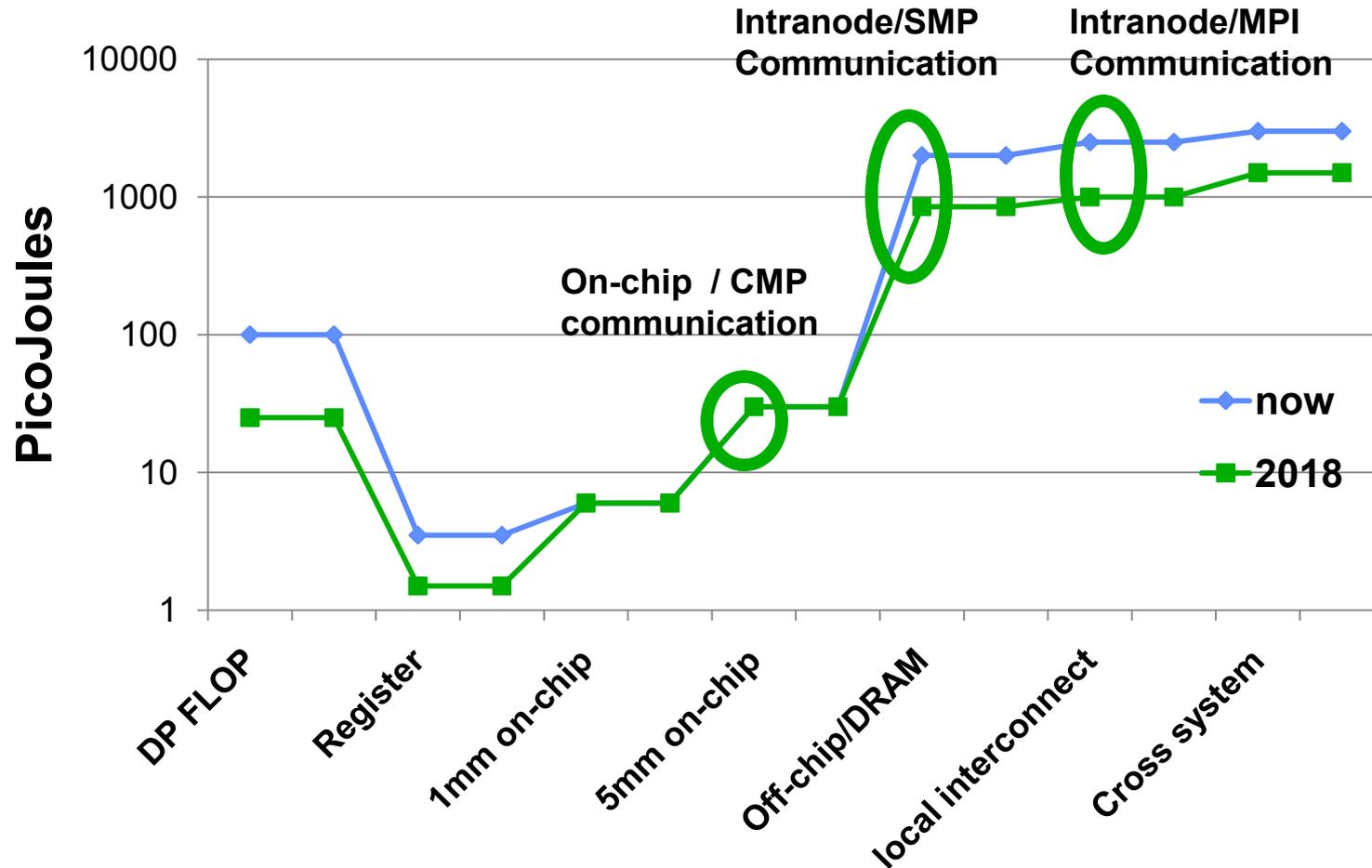


■ Dollars/Mbyte     ▲ Dollars/MFLOP  
 The cost to sense, collect, generate and calculate data is declining much faster than the cost to access, manage and store it

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

# Where the Energy Goes

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



# Summary of Hardware Trends

---

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

- **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 and Hybrid Programming

- **Coarray Fortran (CAF)**
  - Compilers from Cray, Rice and PGI (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

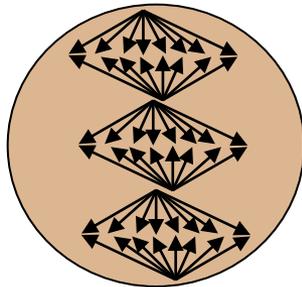
## DARPA High Productivity Computer Systems (HPCS) language project:

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

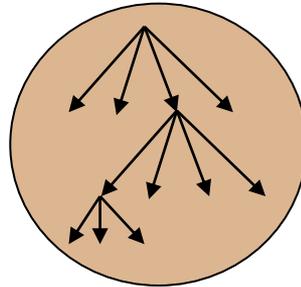
# Two Parallel Language Questions

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

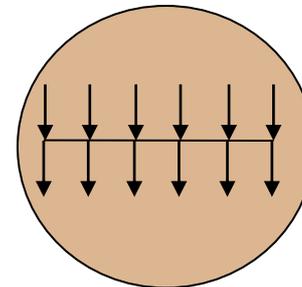
## • What is the parallel control model?



**data parallel**  
**(single thread of control)**

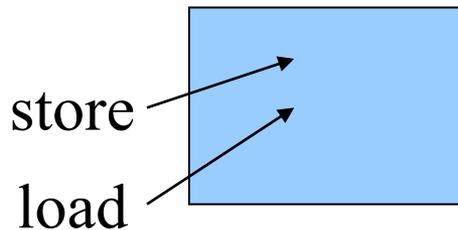


**dynamic**  
**threads**

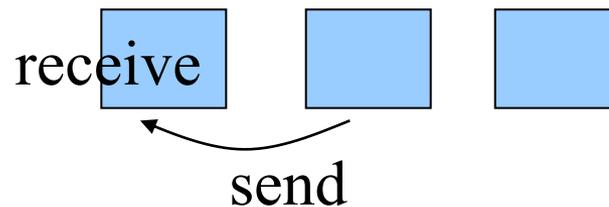


**single program**  
**multiple data (SPMD)**

## • What is the model for sharing/communication?



**shared memory**



**message passing**

**implied synchronization for message passing, not shared memory**

# SPMD Execution Model

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

- **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**
- **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

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

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

➔ Data definition

**!HPF\$ DISTRIBUTE A(block,block), B(...)**

do j = 2, m-1

➔ Loop over y-dimension

do i = 2, n-1

➔ Vectorizable loop over x-dimension

B(i,j) = ... A(i,j)

➔ Calculate B,

... A(i-1,j) ... A(i+1,j)

➔ using upper and lower,  
➔ left and right value of A

... A(i,j-1) ... A(i,j+1)

end do

end do

- 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

# Dynamic Tasking - Cilk

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

```
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*

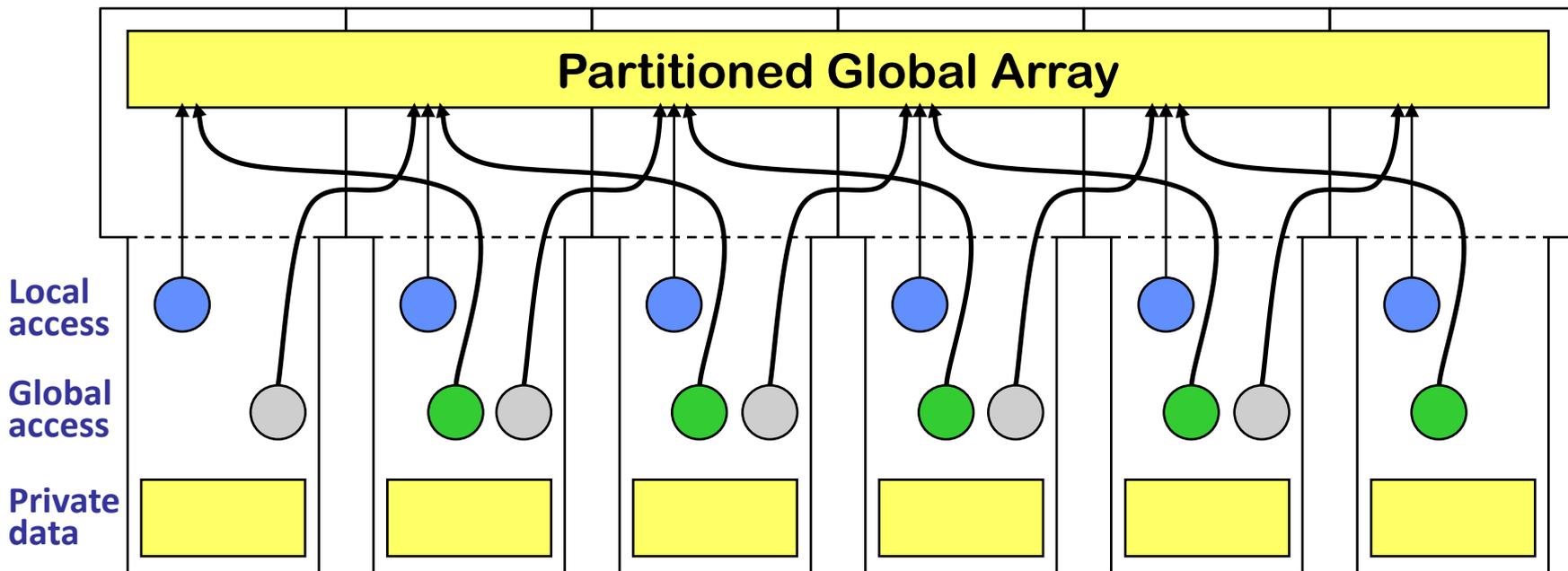
- Task parallel languages are typically implemented with shared memory
- No explicit control over locality; runtime system will schedule related tasks nearby or on the same core
- The HPCS languages support these in a PGAS memory model which yields an interesting and challenging runtime problem

# Partitioned Global Address Space (PGAS) Languages

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

- Defining PGAS principles:

- 1) The *Global Address Space* memory model allows any thread to read or write memory anywhere in the system
- 2) It is *Partitioned* to indicate that some data is local, whereas other data is further away (slower to access)



# Two Concepts in the Memory Space

---

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

- **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: 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

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

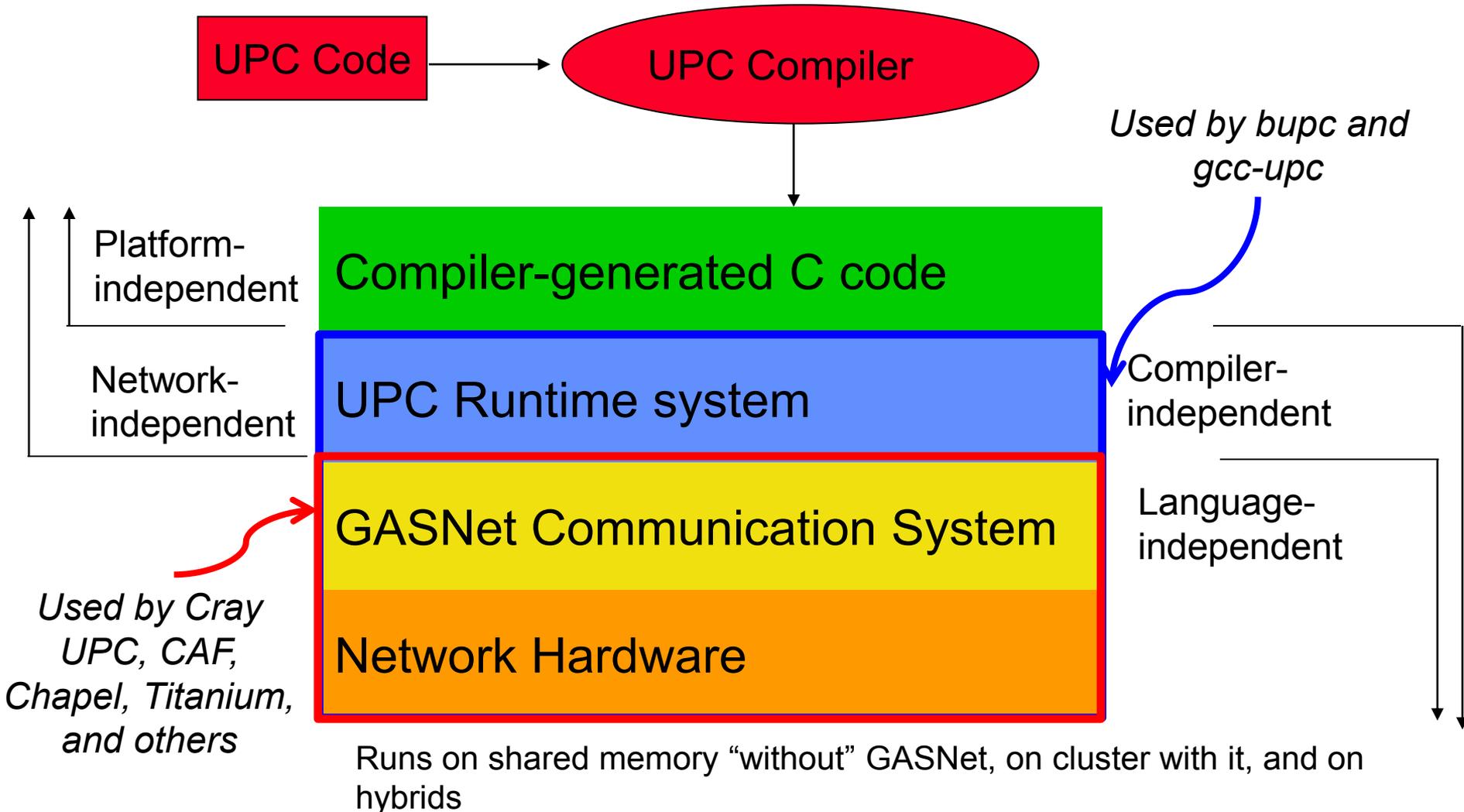
- **Message Passing Interface (MPI)**
  - Library with message passing routines
  - Unforced locality control through separate address spaces
- **OpenMP**
  - Language extensions with shared memory worksharing directives
  - Allows shared data structures without locality control



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

# Understanding Runtime Behavior - Berkeley UPC Compiler

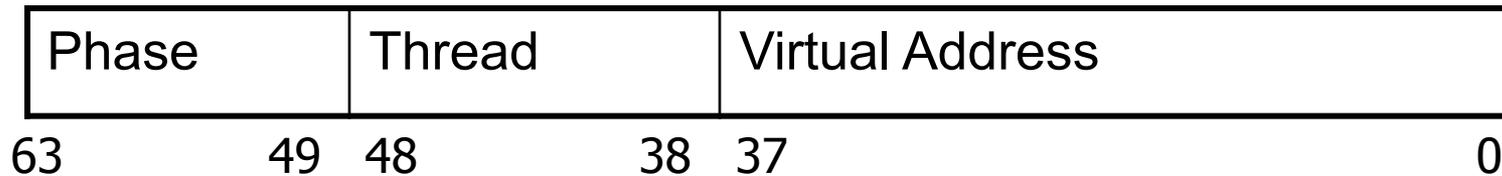
- Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Applications and Hybrid Programming



# UPC Pointers

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

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block) so that operations like ++ move through the array correctly
- Example implementation



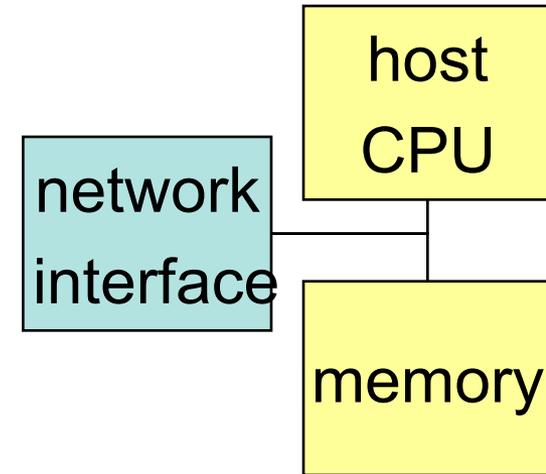
# One-Sided vs Two-Sided Communication

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

one-sided put message



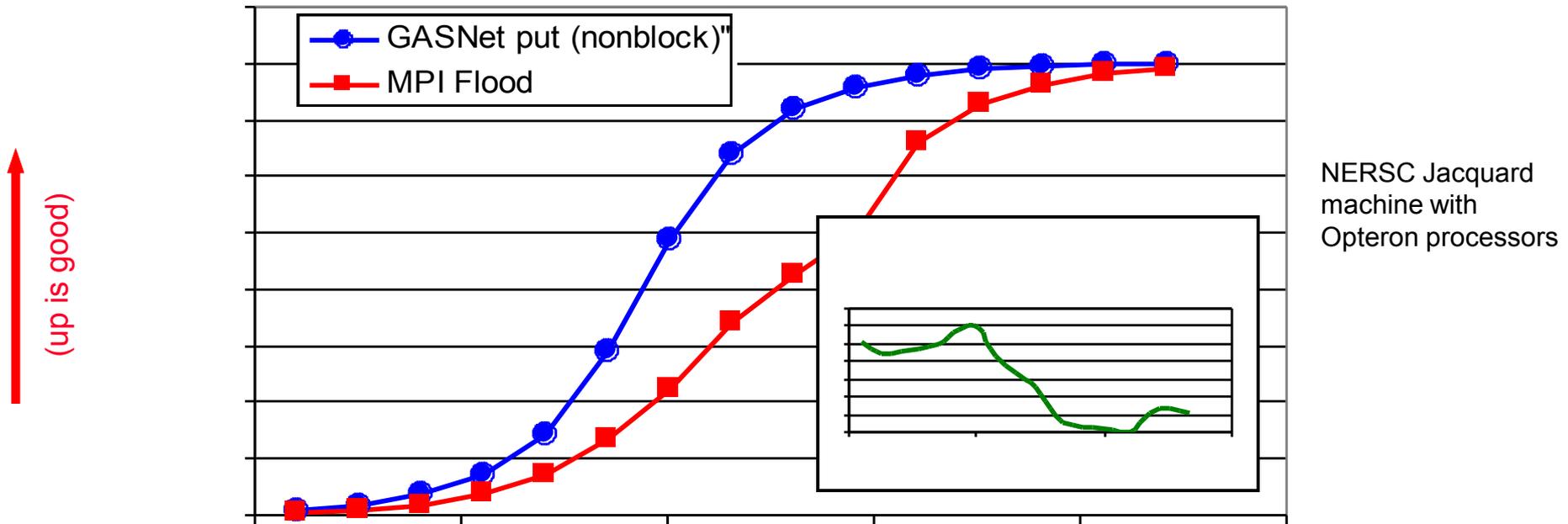
two-sided message



- **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

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

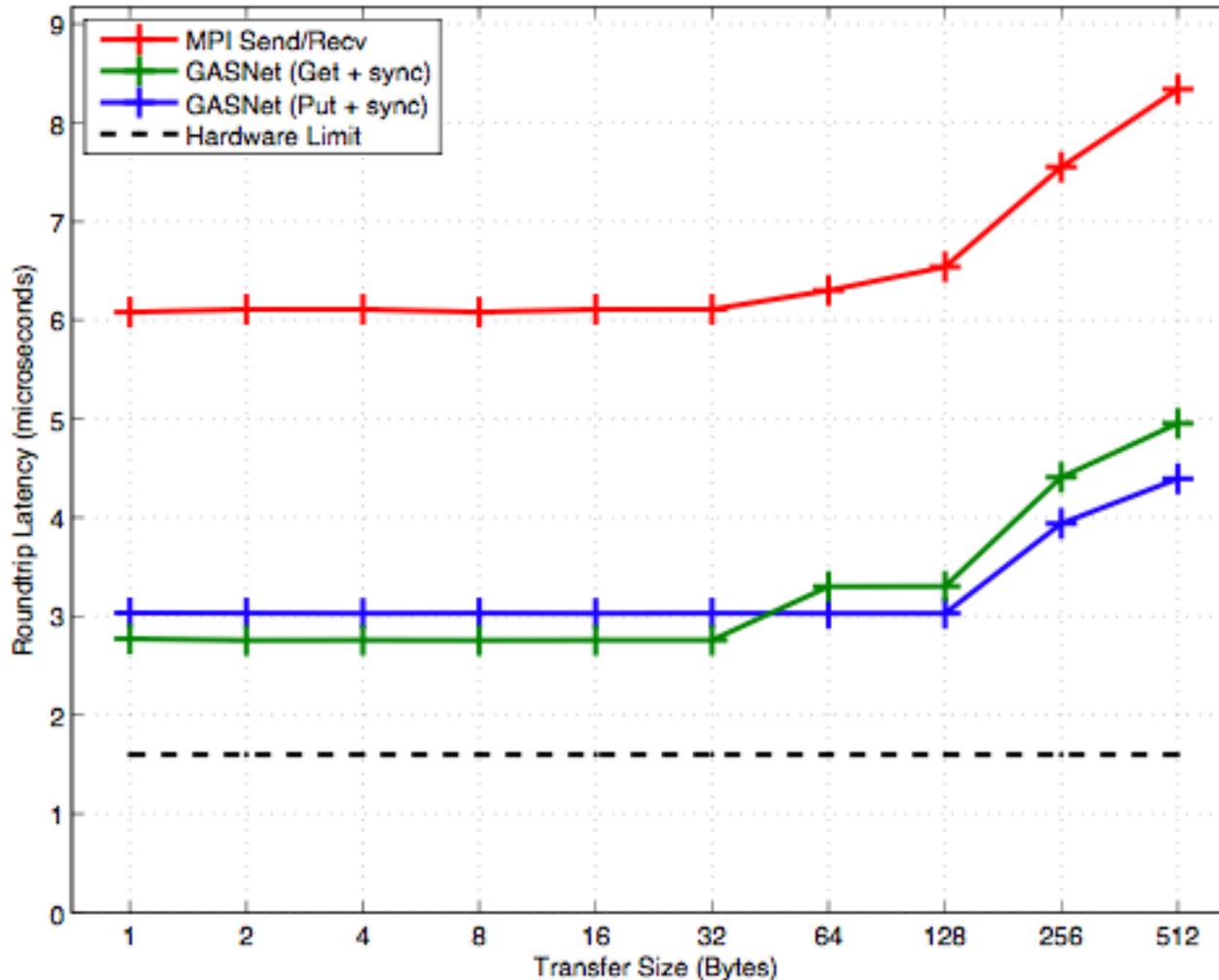


- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point ( $N^{1/2}$ ) differs by *one order of magnitude*
- This is not a criticism of the implementation!

Joint work with Paul Hargrove and Dan Bonachea

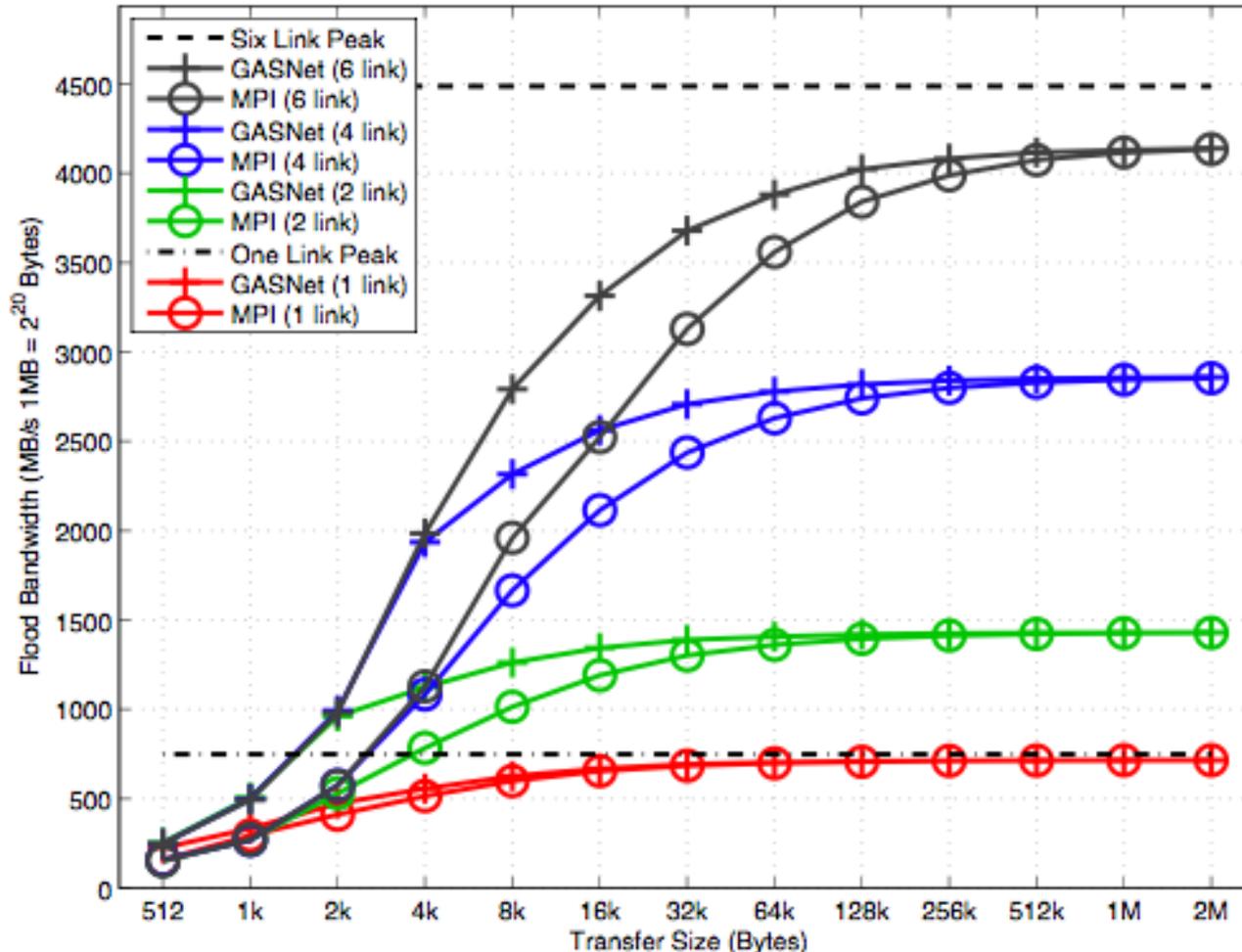
# GASNet vs MPI Latency on BG/P

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



# GASNet vs. MPI Bandwidth on BG/P

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



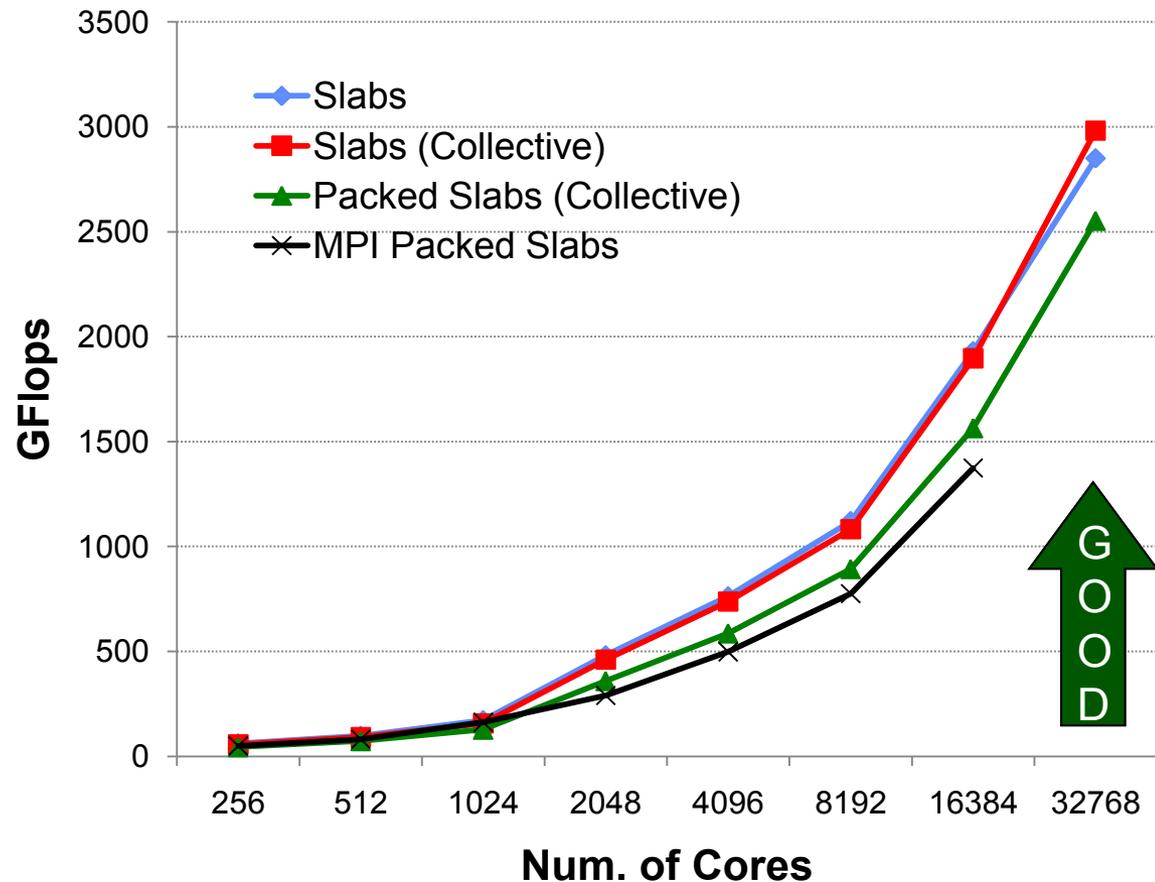
- **GASNet outperforms MPI on small to medium messages, especially when multiple links are used.**

# FFT Performance on BlueGene/P

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

- 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

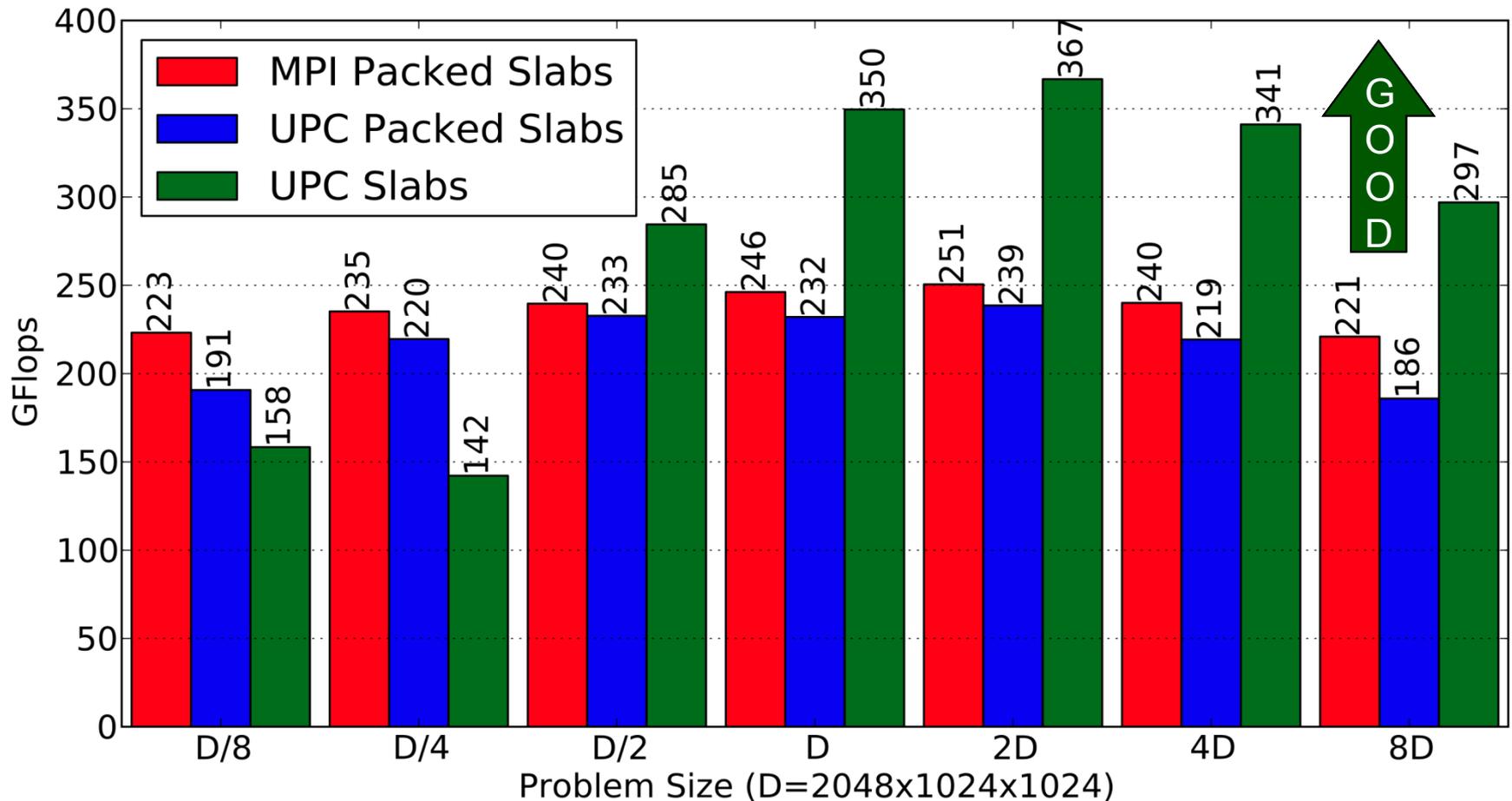
HPC Challenge Peak as of July 09 is  
~4.5 Tflops on 128k Cores



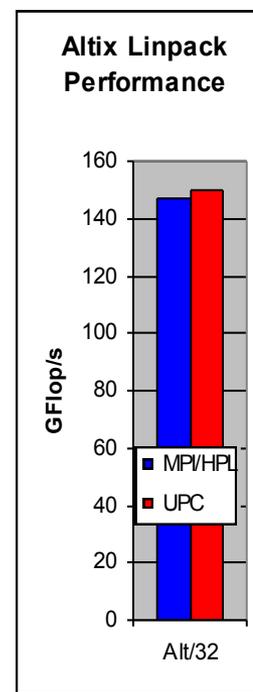
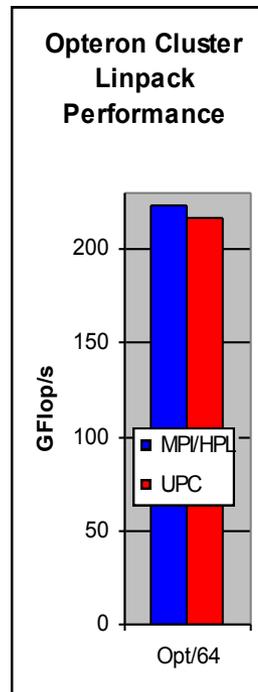
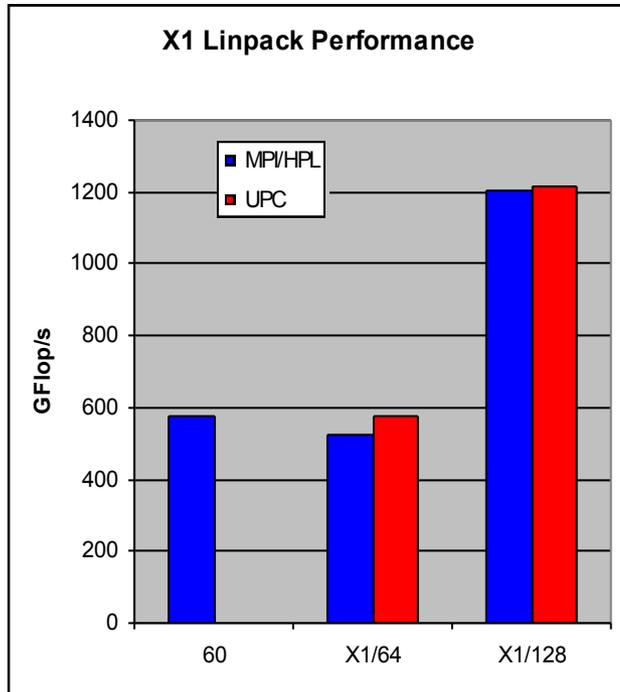
# FFT Performance on Cray XT4

- 1024 Cores of the Cray XT4

- Uses FFTW for local FFTs
- Larger the problem size the more effective the overlap



# UPC HPL 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://en.wikipedia.org/wiki/Unified_Parallel_C)
- <http://upc.gwu.edu/>
- <https://upc-wiki.lbl.gov/UPC/>
- <http://upc.gwu.edu/documentation.html>
- <http://upc.gwu.edu/download.html>

→ Main UPC homepage

→ UPC wiki

→ Language specs

→ UPC compilers

- **CAF**

- [http://en.wikipedia.org/wiki/Co-array\\_Fortran](http://en.wikipedia.org/wiki/Co-array_Fortran)
- <http://www.co-array.org/>
- Part of upcoming Fortran 2008
- <http://www.g95.org/coarray.shtml>

→ Main CAF homepage

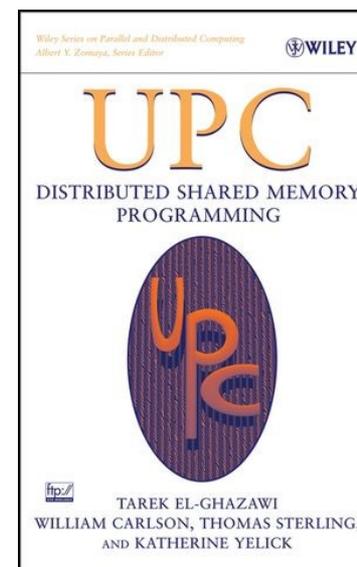
→ g95 compiler

skipped

# UPC

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

- **UPC Language Specification (V 1.2)**
  - The UPC Consortium, June 2005
  - [http://upc.gwu.edu/docs/upc\\_specs\\_1.2.pdf](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



- From <http://www.nag.co.uk/SC22WG5/>
- 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>

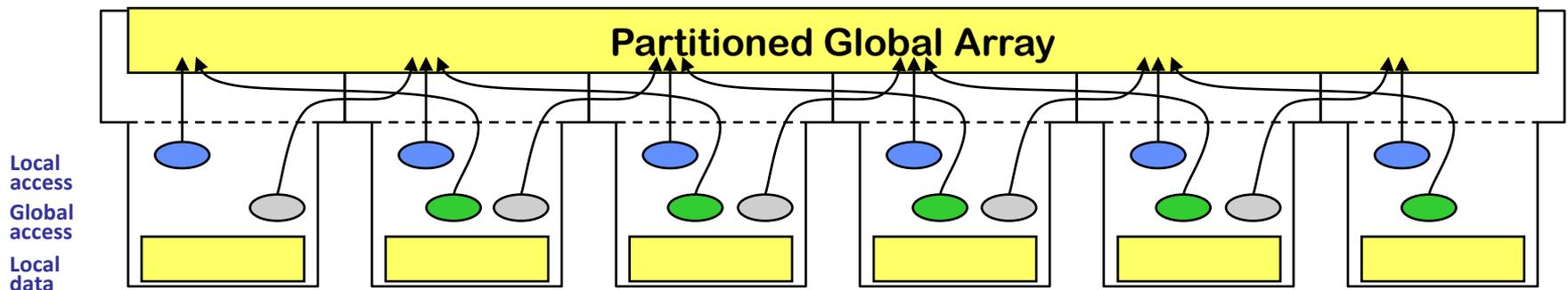
#### Older versions:

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

# Programming styles with PGAS

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

- **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



# Coming from MPI – what's different with PGAS?

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

```
size = num_images()
myrank = this_image() - 1
```

```
m1 = (m+size-1)/size; ja=1; je= m1; ! Same values on all processes
jax=ja-1; jex=je+1 // extended boundary with halo
```

```
ja_loop=1; if(myrank==0) jaloop=2; jeloop=min((myrank+1)*m1,m-1) - myrank*m1;
```

```
Real :: A(n, jax:jex), B(n, jax:jex)
```

```
do j = jaloop, jeloop ! Orig.: 2, m-1
```

```
do i = 2, n-1
```

```
  B(i,j) = ... A(i,j)
           ... A(i-1,j) ... A(i+1,j)
           ... A(i,j-1) ... A(i,j+1)
```

```
end do
```

```
end do
```

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

in original index range

remove range of lower processes

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

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

# Irregular Applications

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

- **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:
    - Not statically load balanced (even with graph partitioning, etc.)
    - Some kind of dynamic load balancing needed with 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 is an add-on

# Distributed Tasking API for UPC

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

// enqueue a task into the distributed queue
int taskq_put(upc_taskq_t *, upc_task_t*);

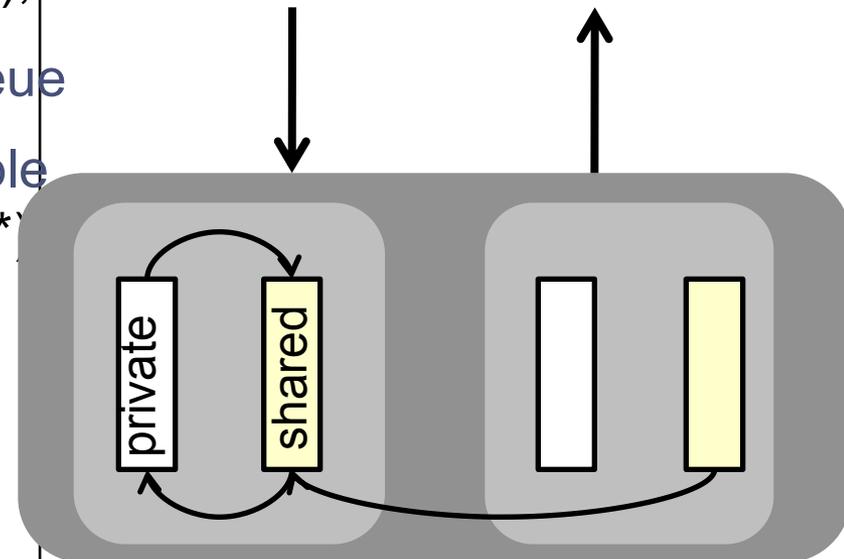
// dequeue a task from the local task queue
// returns null if task is not readily available
int taskq_get(upc_taskq_t *, upc_task_t *)

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

// free distributed task queue memory
int taskq_free(shared bupc_taskq_t *);
```

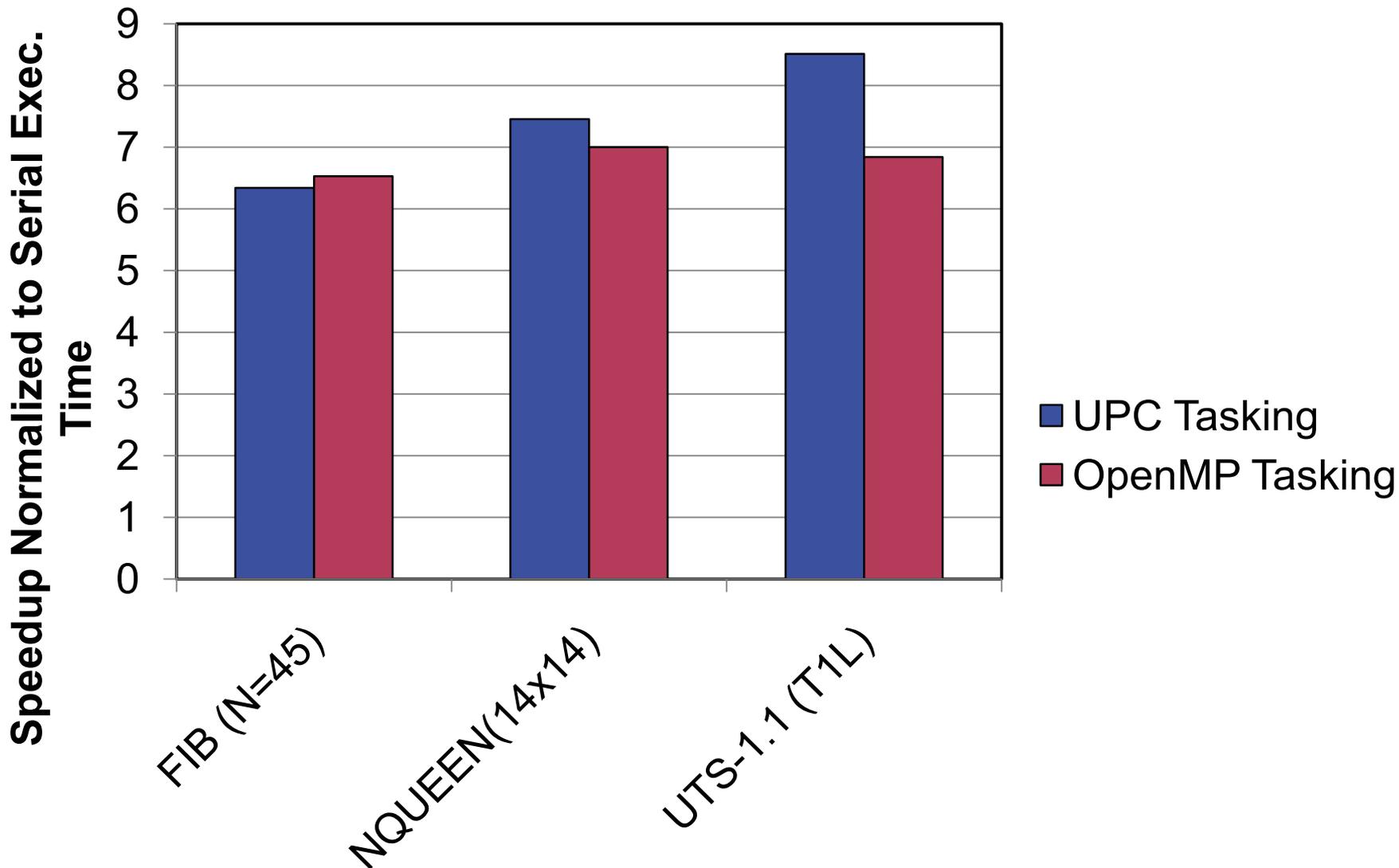
*internals are hidden from user, except that dequeue operations may fail and provide hint to steal*

enqueue      dequeue



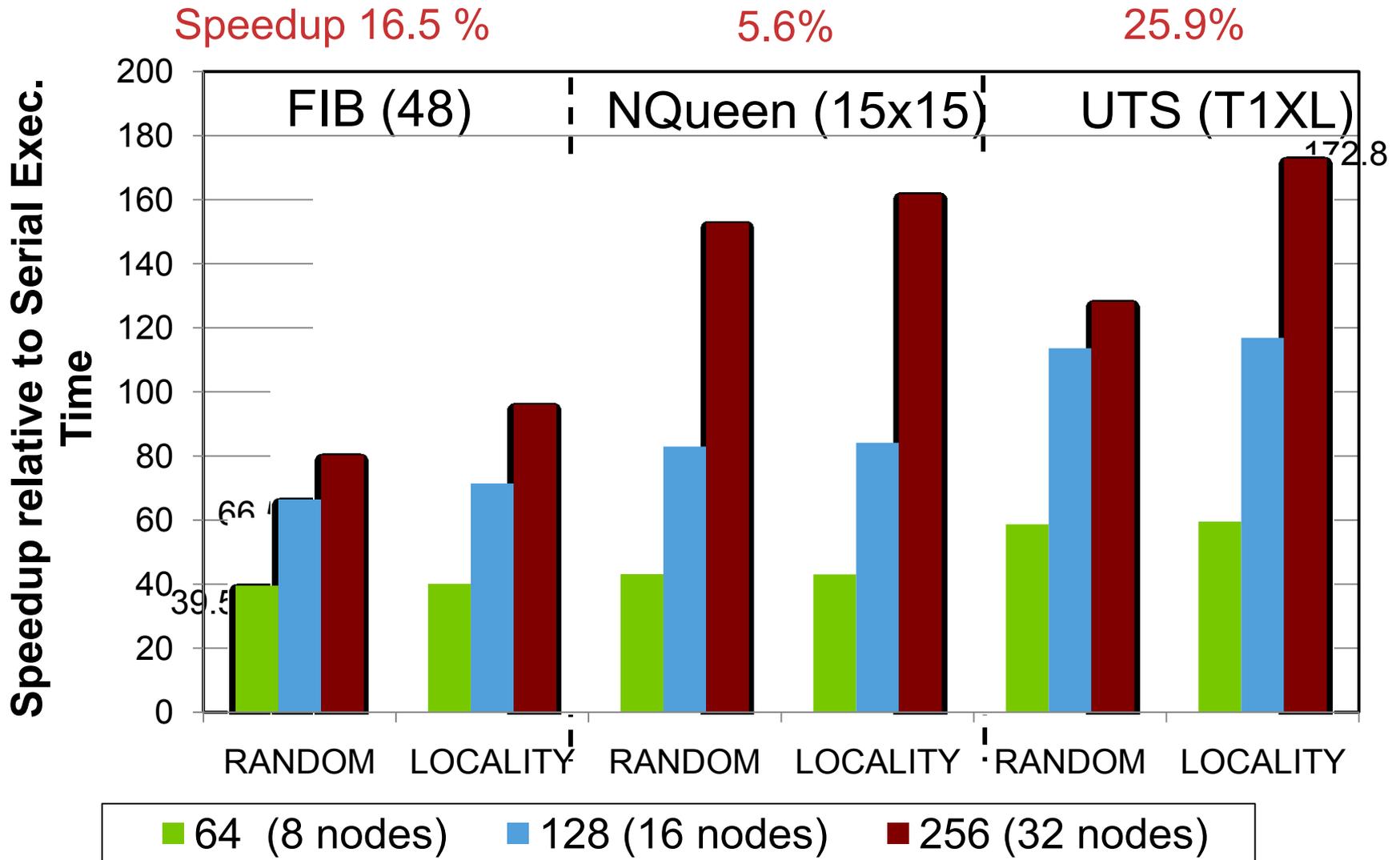
# UPC Tasking on Nehalem 8 core SMP

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



# Multi-Core Cluster Performance.

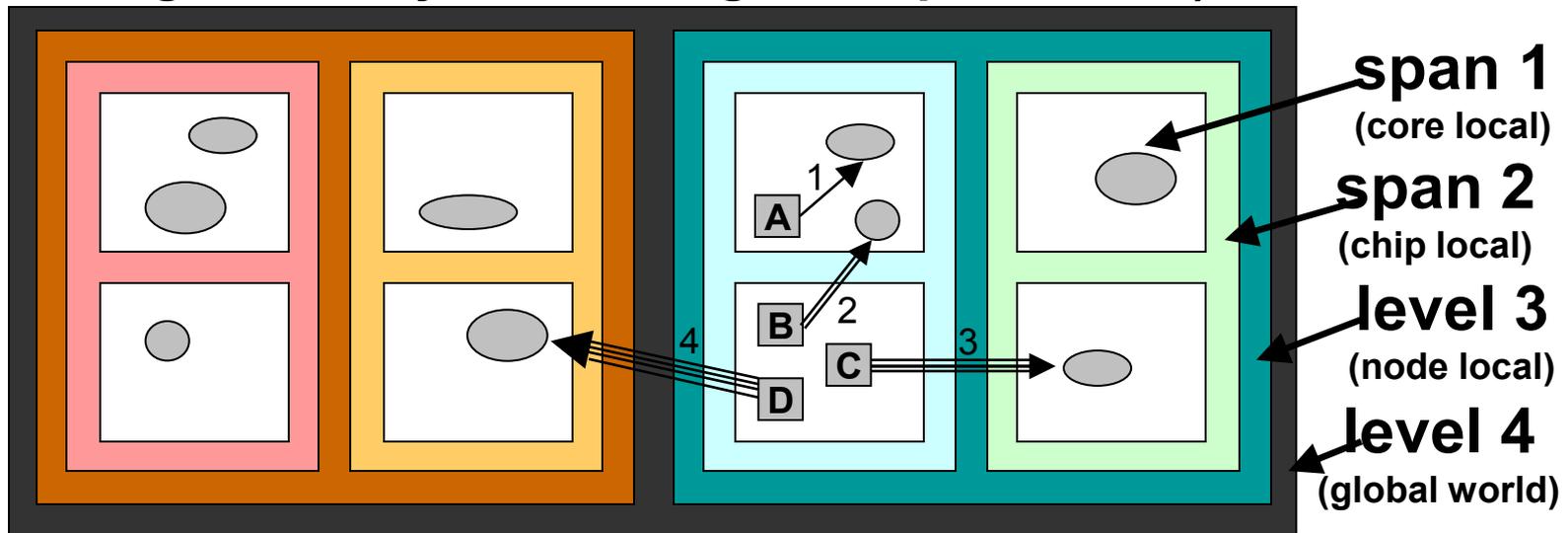
- Basic PGAS concepts
  - Trends
  - UPC and CAF basic syntax
  - Advanced synchronization concepts
  - Applications and Hybrid Programming



# Hierarchical PGAS Model

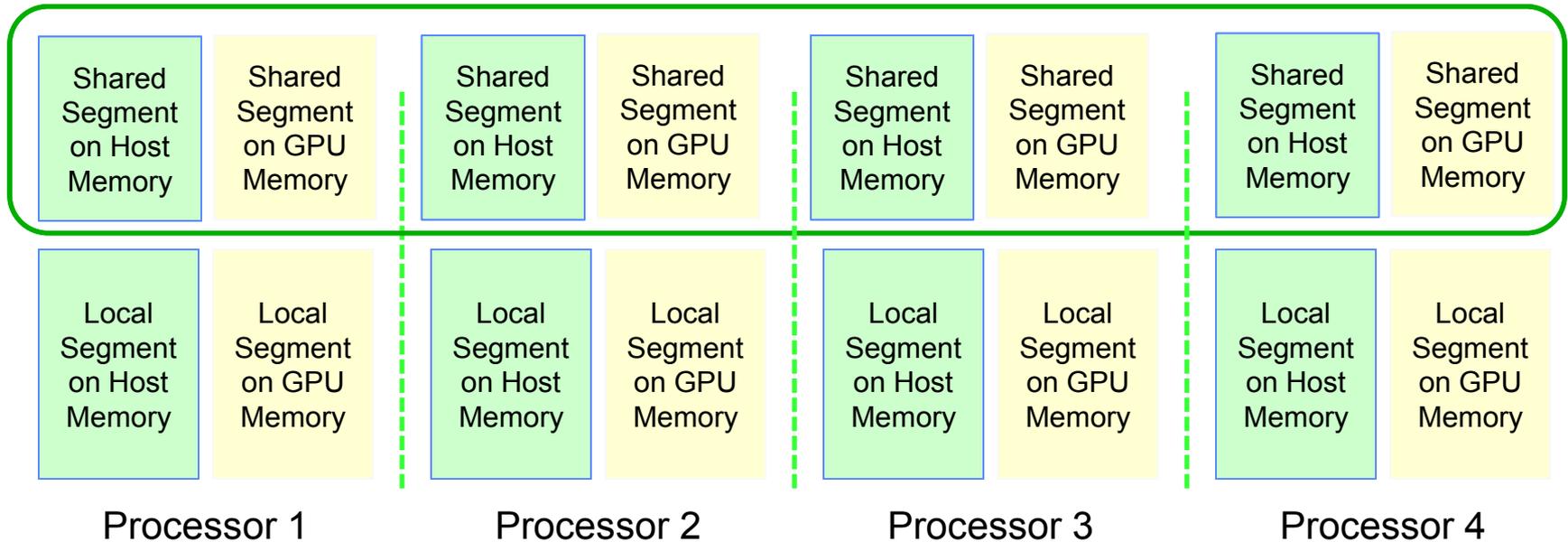
- **Basic PGAS concepts**
  - Trends
  - UPC and CAF basic syntax
  - Advanced synchronization concepts
  - Applications and Hybrid Programming

- 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)



# Hybrid Partitioned Global Address Space

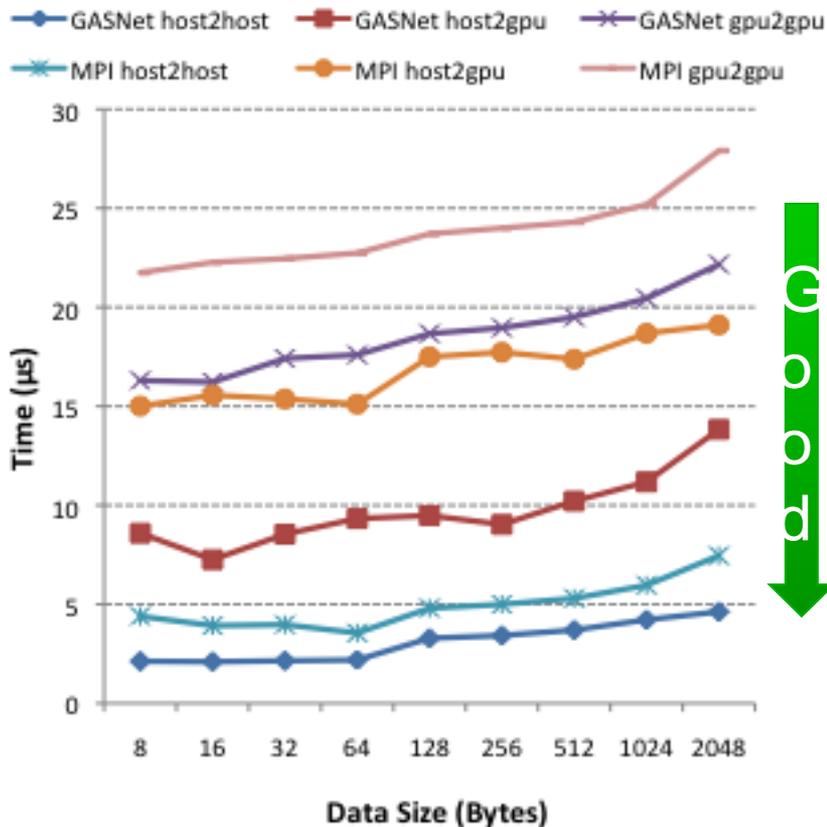
- Basic PGAS concepts
  - Trends
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Applications and Hybrid Programming



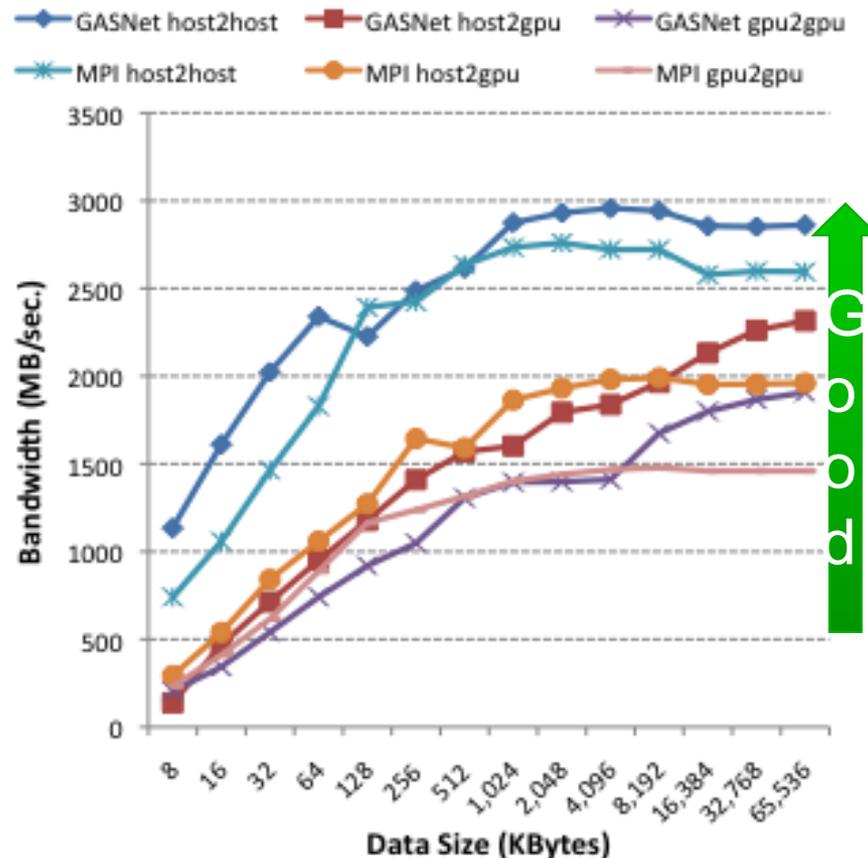
- ❖ Each thread has only two shared segments, which can be either in host memory or in GPU memory, but not both.
- ❖ Decouple the memory model from execution models; therefore it supports various execution models.
- ❖ Caveat: type system and therefore interfaces blow up with different parts of address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

# GASNet GPU Extension Performance

## Latency



## Bandwidth



# Compilation and Execution

- **Basic PGAS concepts**
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming

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

see also  
UPC-pgi

# Compilation and Execution

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

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

see also  
Cray UPC

see also  
Cray Fortran

# First exercise

- Basic PGAS concepts
  - Exercises
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming

- **Purpose:**
    - get acquainted with use of compiler and run time environment
    - use basic intrinsics
    - first attempt at data transfer
  - **Copy skeleton program to your working directory:**
    - `cp ../skel/hello_serial.f90 hello_caf.f90`
    - `cp ../skel/hello_serial.c hello_upc.c`
  - **Add statements**
    - UPC: also include file
- to enable running on multiple images**
- only one task should write „hello world“

hello

- **Add the following declarations and statements:**

```
integer :: x[*] = 0
:
x = this_image()
if (this_image() > 1) then
    write(*, *) 'x from 1 is ',x[1]
end if
```

Fortran

incorrect. why?

```
shared [] int x[THREADS];
:
x[MYTHREAD] = 1 + MYTHREAD;
if (MYTHREAD > 0) {
    printf("x from 0 is %i\n",
        x[0]);
}
```

C

incorrect. why?

- and observe what happens if run repeatedly with more than one image/thread

# 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

# Partitioned Global Address Space: Distributed variable

- Basic PGAS concepts
  - UPC and CAF basic syntax
    - Shared entities
  - Advanced synchronization concepts
  - Hybrid Programming

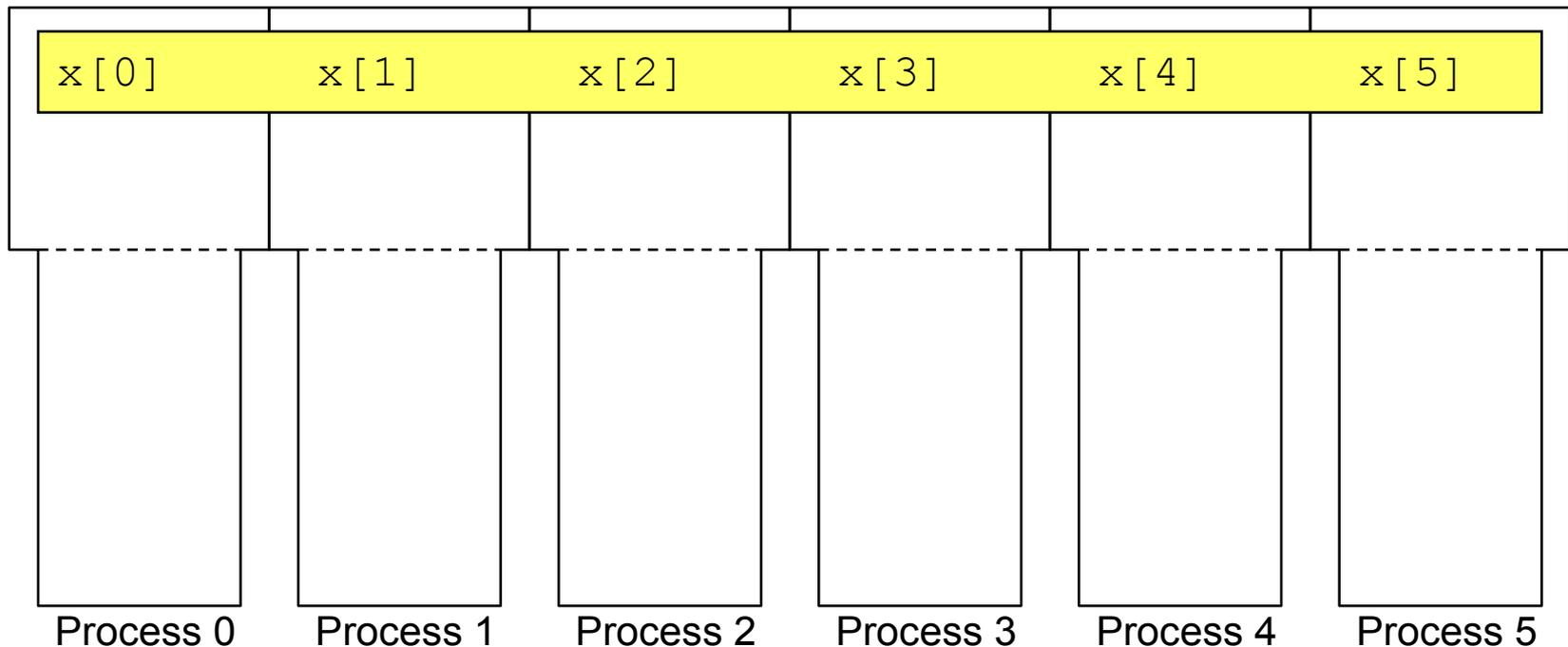
- **Declaration:**

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

UPC: "Parallel dimension"

- **Data distribution:**

CAF: "Codimension"



# Partitioned Global Address Space: Distributed array

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

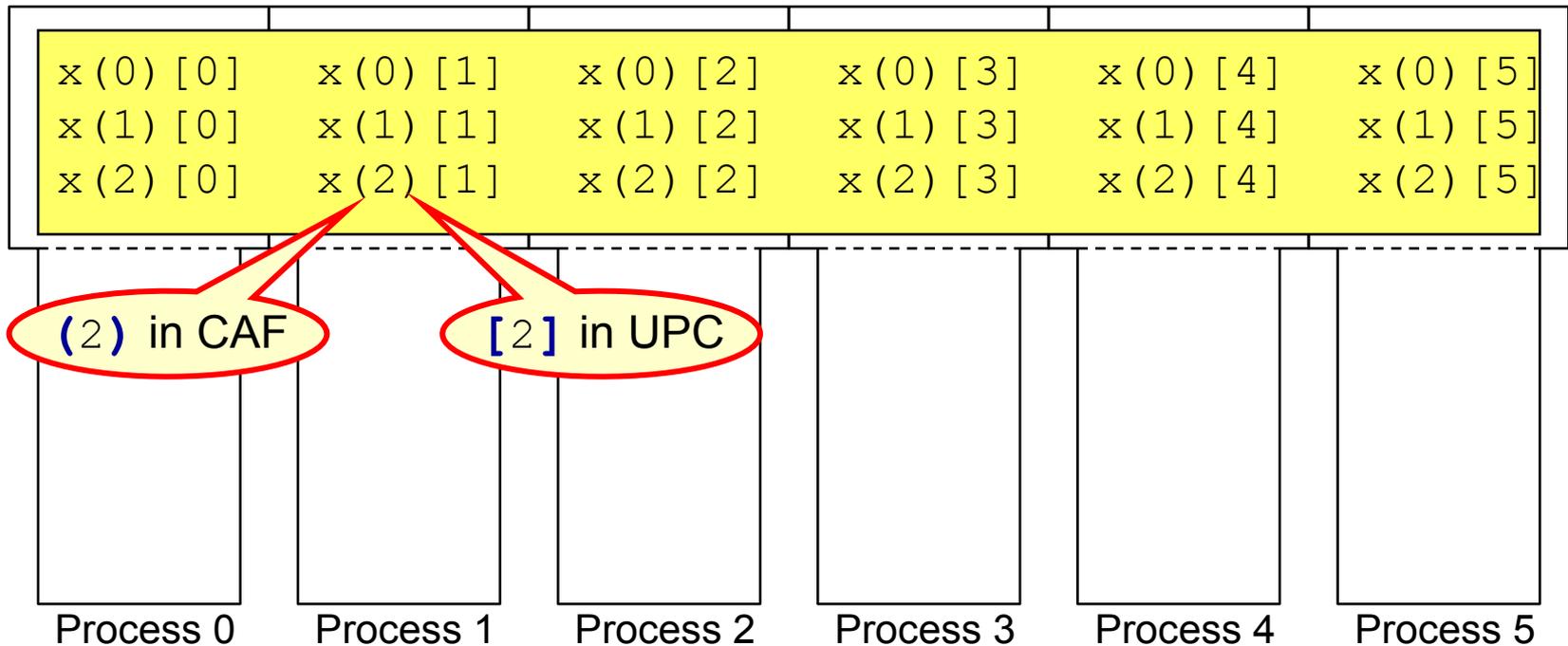
- **Declaration:**

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

UPC: "Parallel dimension"

CAF: "Codimension"

- **Data distribution:**



# Distributed arrays with UPC

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

- UPC shared objects must be statically allocated
- Definition of shared data:
  - **shared** [**blocksize**] type variable\_name;
  - **shared** [**blocksize**] type array\_name[dim1];
  - **shared** [**blocksize**] type array\_name[dim1][dim2];
  - ...

the dimensions  
define which  
elements exist

- Default: blocksize=1
- { The distribution is always round robin with chunks of **blocksize** elements
- { Blocked distribution is implied if last dimension==THREADS and blocksize==1

See next slides

# UPC shared data – examples

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

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

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]

Thread 0    Thread 1    Thread 2    Thread 3

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

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]

Thread 0    Thread 1    Thread 2    Thread 3

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

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]

Thread 0    Thread 1    Thread 2    Thread 3

THREADS=1<sup>st</sup> dim!    identical at compile time

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

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]

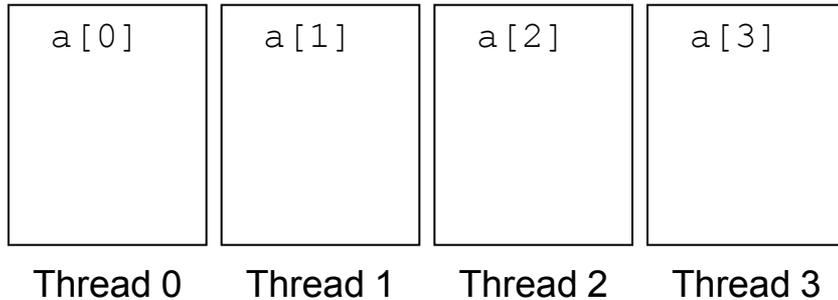
Thread 0    Thread 1    Thread 2    Thread 3

Courtesy of Andrew Johnson

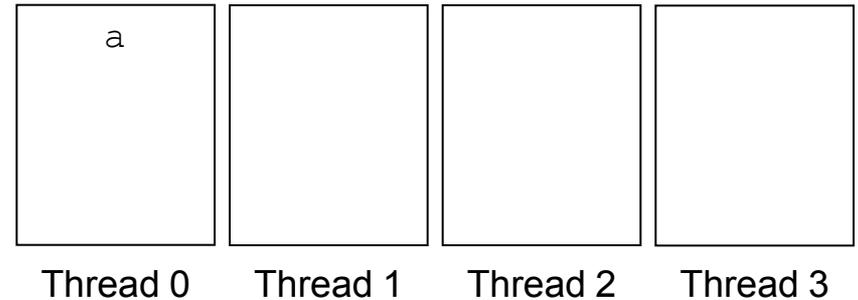
# UPC shared data – examples (continued)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

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

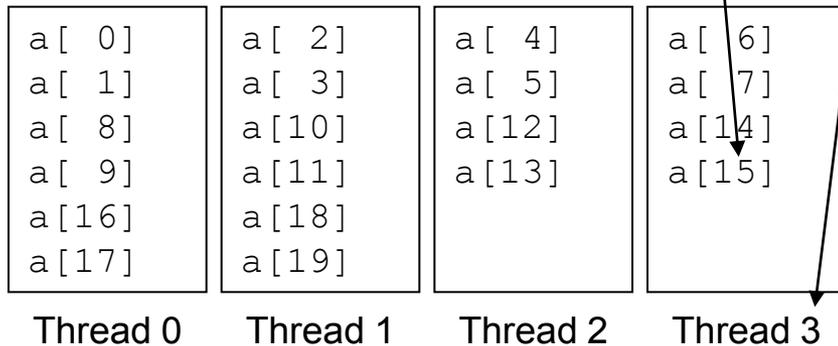


```
shared float a;
// located only in thread 0
```



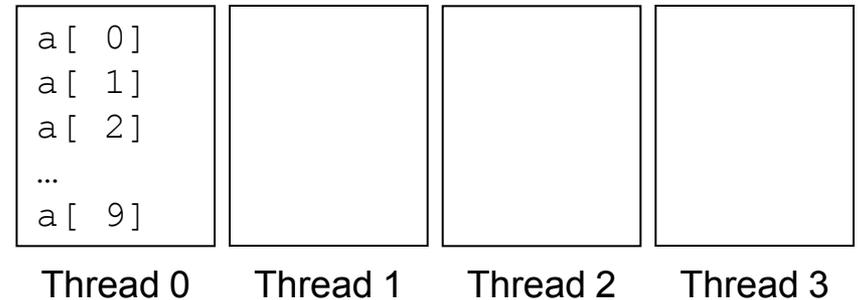
```
shared [2] float a[20];
```

upc\_threadof(&a[15]) == 3



Blank blocksize → located only in thread 0

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



Courtesy of Andrew Johnson

# Integration of the type system (static type components)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

## • 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;
```

enforced storage order

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

components „lifted“ to shared area

- 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

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

- **UPC:**

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

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

*x now equals x[MYTHREAD]
```

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

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

x now equals x[myrank]
```

- (1-based ranks)**

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

x now equals x[myrank]
```

# CAF-only: Multidimensional coindexing

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

- 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

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

- **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, (/ 3, 2 /))
```

image on which y[3, 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)

- Basic PGAS concepts
  - UPC and CAF basic syntax
    - Shared entities
- Advanced synchronization concepts
- Hybrid Programming

- **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

$a_1, \dots, a_N$

- blocked distribution:

$a_1, \dots, a_b$

$a_{b+1}, \dots, a_{2b}$

$\dots, a_N$

(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=1, nlocal
  j = ... ! global index
  a(i) = ...
end do
```

may vary between images

- **UPC: `upc_forall`**

- integrates affinity with loop construct

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

affinity expression

- **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] = ... ;
  }
}
```

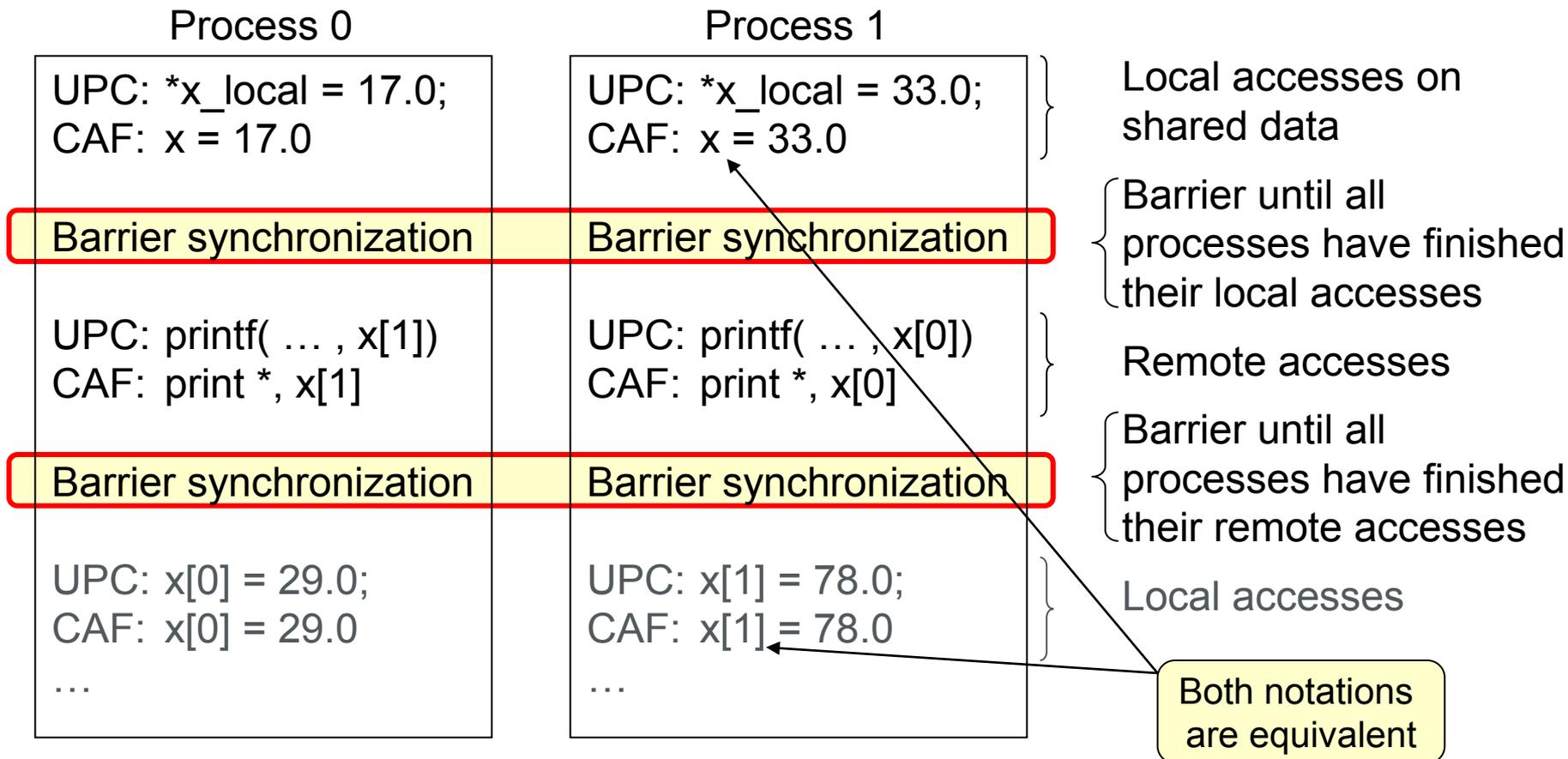
- conditional may be inefficient
- cyclic distribution may be slow

- affinity expression:
  - an integer  $\rightarrow$  execute if `i%THREADS == MYTHREAD`
  - a global address  $\rightarrow$  execute if `upc_threadof(...) == MYTHREAD`
  - `continue` or empty  $\rightarrow$  all threads (use for nested `upc_forall`)
- example above: could replace „i“ with „&a[i]“

# Typical collective execution with access epochs

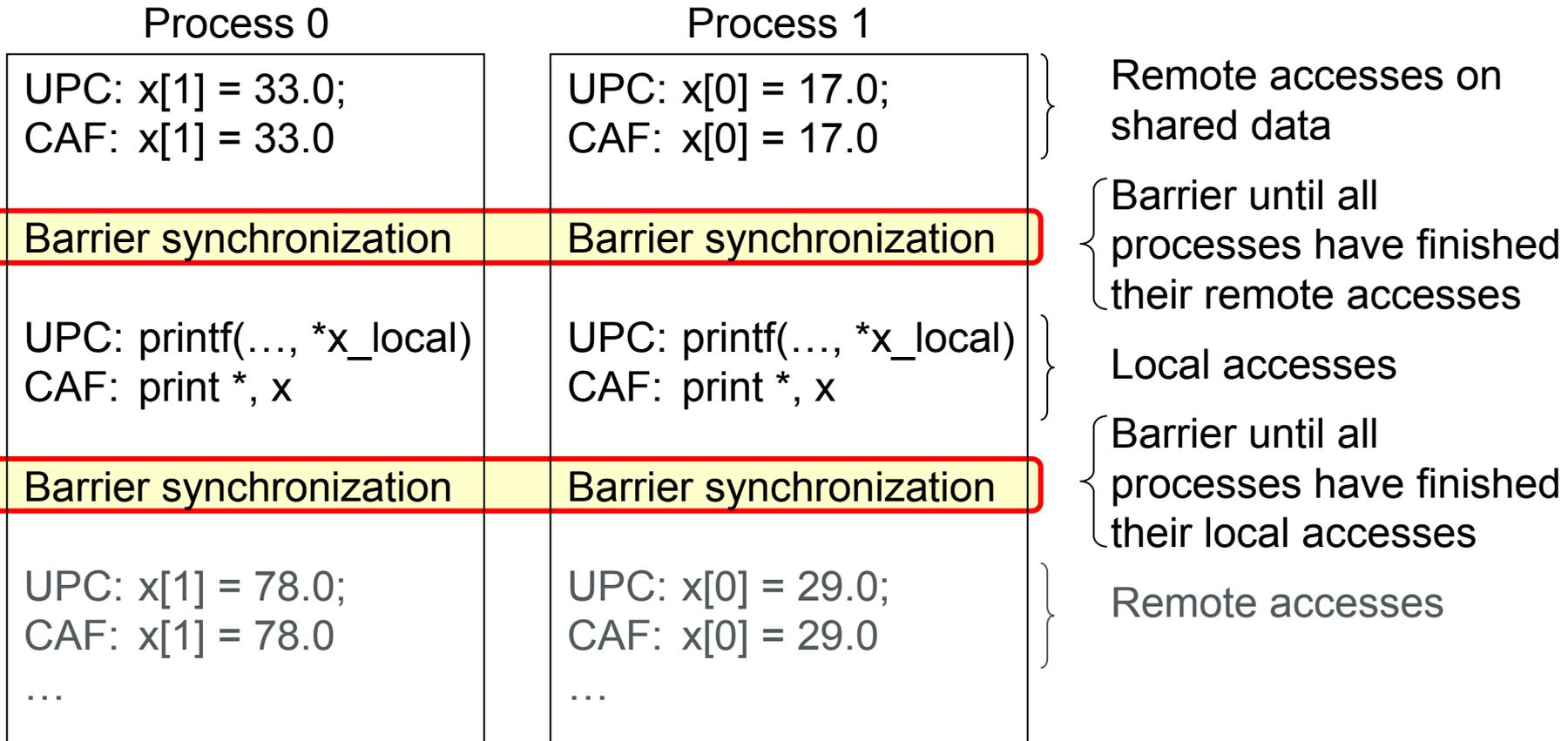
- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming

(CAF: segments)



# Collective execution – same with remote write / local read

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



# Synchronization

- Basic PGAS concepts
- **UPC and CAF basic syntax**
- Advanced synchronization concepts
- Hybrid Programming

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

**Otherwise  
race condition!**

- UPC:

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

- CAF:

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

# Examples

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

- UPC:

write

```
shared float x[THREADS];  
x[MYTHREAD] = 1000.0 + MYTHREAD;
```

sync

```
upc_barrier;
```

read

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

- CAF:

write

```
real :: x[0:*]
```

```
integer :: myrank, numprocs
```

```
numprocs=num_images(); myrank =this_image()-1
```

sync

```
x = 1000.0 + myrank
```

```
sync all
```

```
print *, 'myrank=', myrank,
```

```
      'x[neighbor=', mod(myrank+1,numprocs),
```

read

```
      ']=', x[mod(myrank+1,numprocs)]
```

# UPC and CAF Basic Syntax

- o Declaration of shared data / coarrays
- o Intrinsic procedures for handling shared data
  - elementary work sharing
- o Synchronization:
  - motivation – race conditions;
  - rules for access to shared entities by different threads/images
- o **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

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- **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

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- 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 realized
  - **CAF**: only **two** of the combinations allowed, and only in a limited manner  
← aliasing is allowed only to local entities

# Pointers continued ...

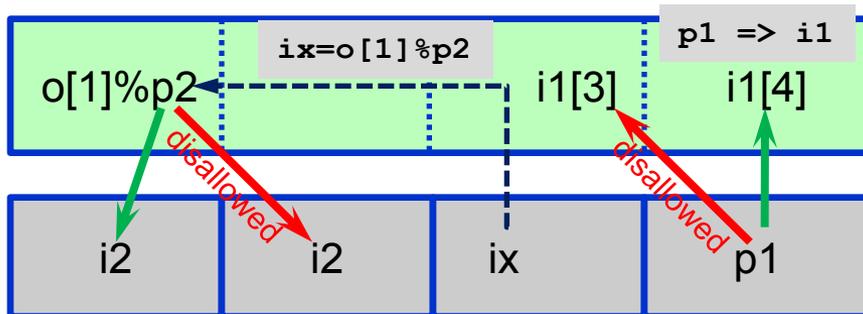
## • CAF:

```
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



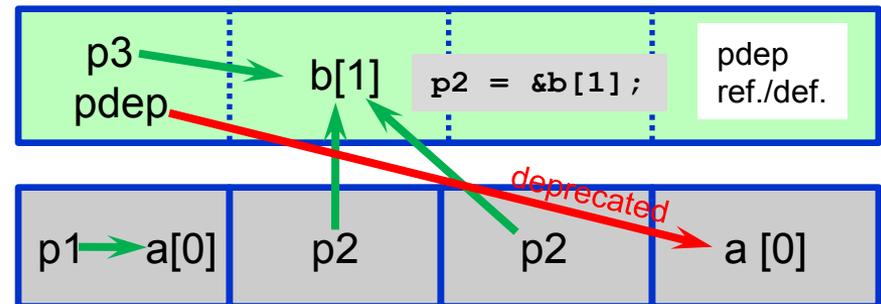
## • UPC:

```
int *p1;
shared int *p2;
shared int *shared p3;
int *shared pdep;

int a[N];
shared int b[N];
```

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

- pointer to shared: addressing overhead



(alias+coindexing) vs. address

# Pointer to local portions of shared data

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- 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

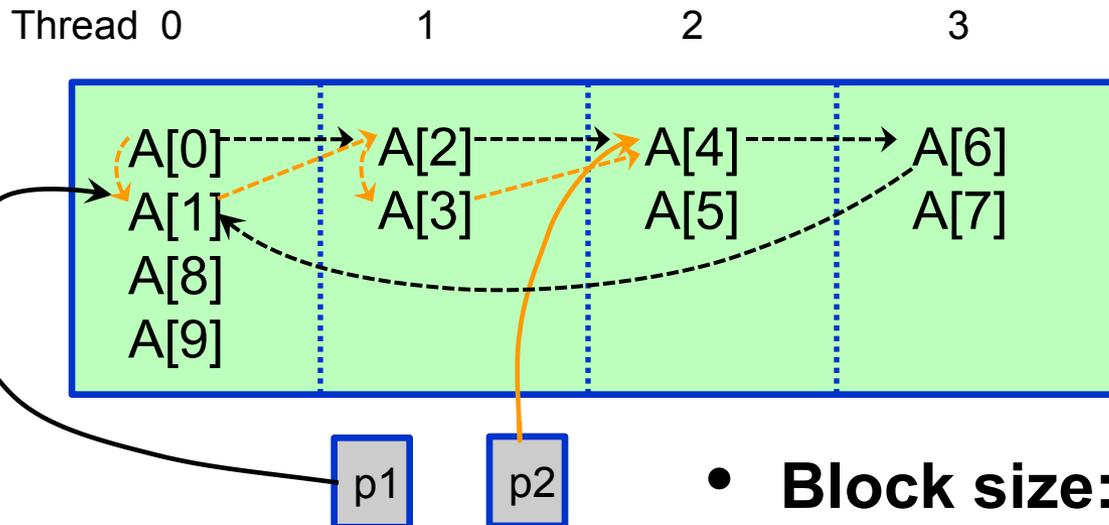
- May have performance advantages

# UPC: Shared Pointer blocking and casting

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- Assume 4 threads:

```
shared [2] int A[10];  
shared int *p1;  
shared [2] int *p2;
```



after pointer increment

```
if (MYTHREAD == 1) {  
    p1 = &A[0]; p2 = &A[0];  
    p1 += 4; p2 += 4;  
}
```

- **Block size:**

- is a property of the shared entity used
- can cast between different block sizes  
→ pointer arithmetic follows blocking („phase“) of pointer!

# UPC dynamic Memory Allocation

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- **upc\_all\_alloc**

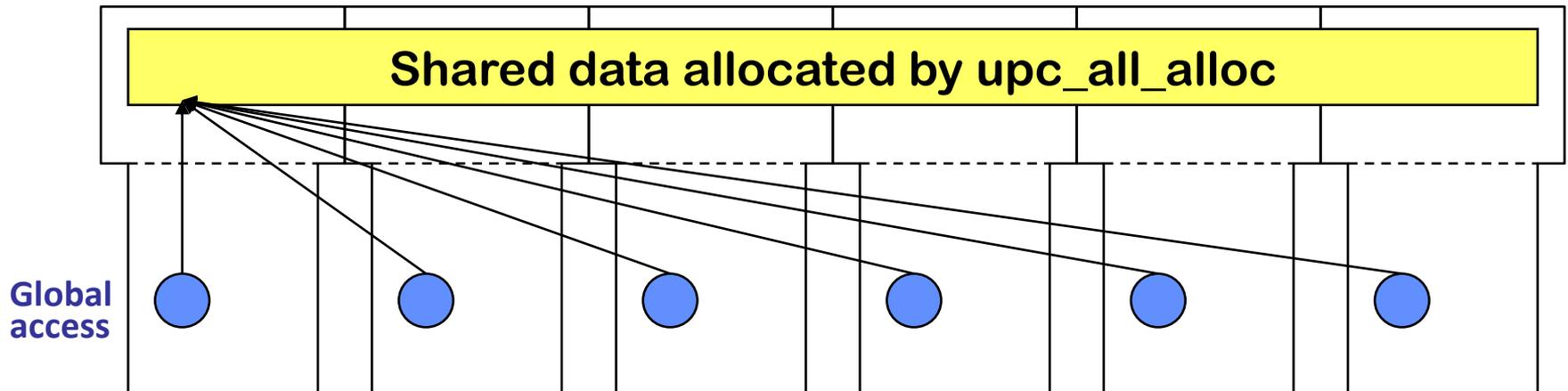
- Collective over all threads (i.e., all threads must call)
- All threads get a copy of the same pointer to shared memory

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

Run time arguments

- Similar result as with static allocation at compile time:

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



# UPC dynamic Memory Allocation (2)

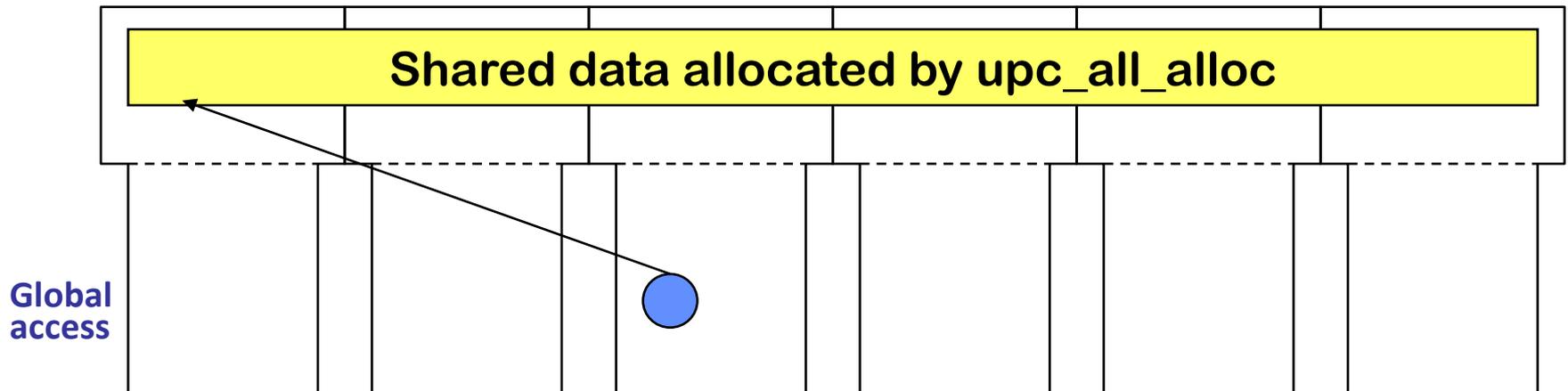
- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- **upc\_global\_alloc**

- Only the calling thread gets a pointer to shared memory

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

- Block size nbytes other than 1 is currently not supported  
(in Berkeley UPC)



# UPC dynamic Memory Allocation (3)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

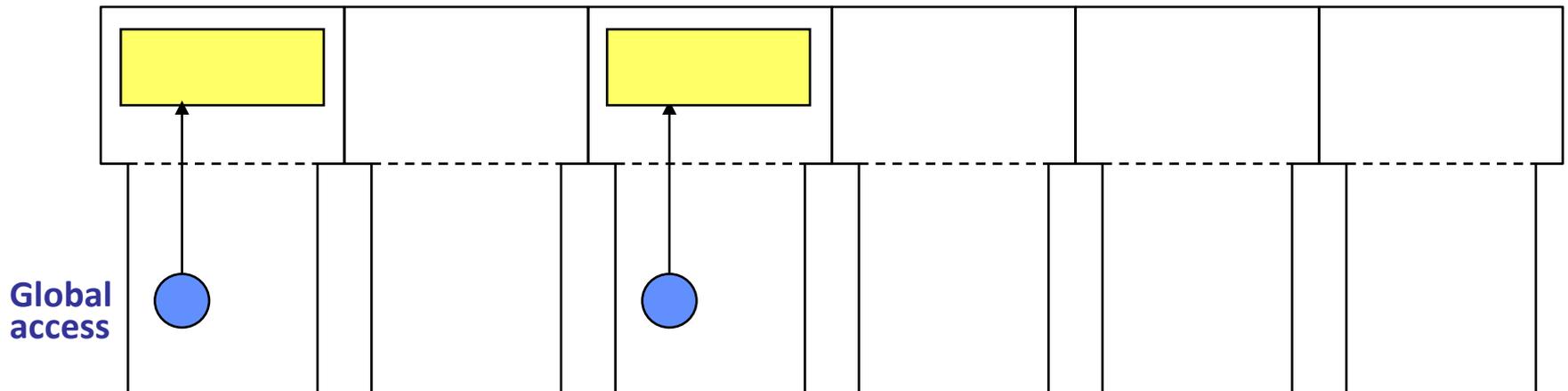
- **upc\_alloc**

- 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
```



# UPC example with dynamic allocation

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

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

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

# UPC example with shared pointers

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

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

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

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

# Integration of the type system

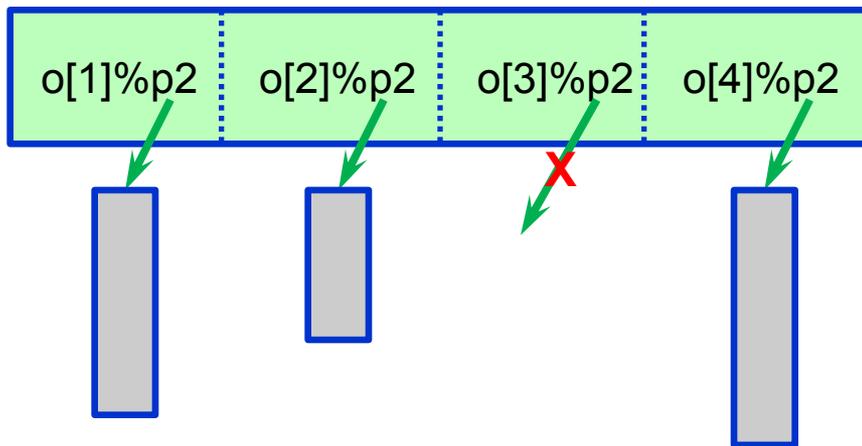
## CAF dynamic components

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

### • 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

```
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
```

remember earlier type definition

or  
o%p2 => var

sz same on each image?

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

# Integration of the type system

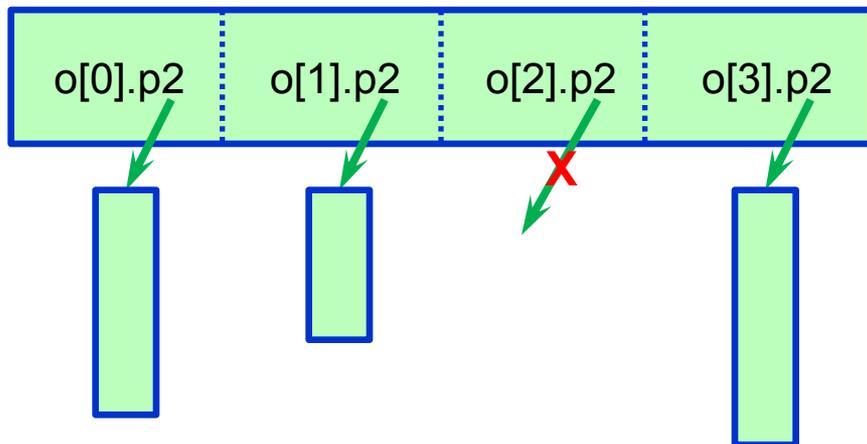
## UPC pointer components

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

### • Type definition

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

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



- avoid undefined results when transferring data between threads

### • Similar step-by-step:

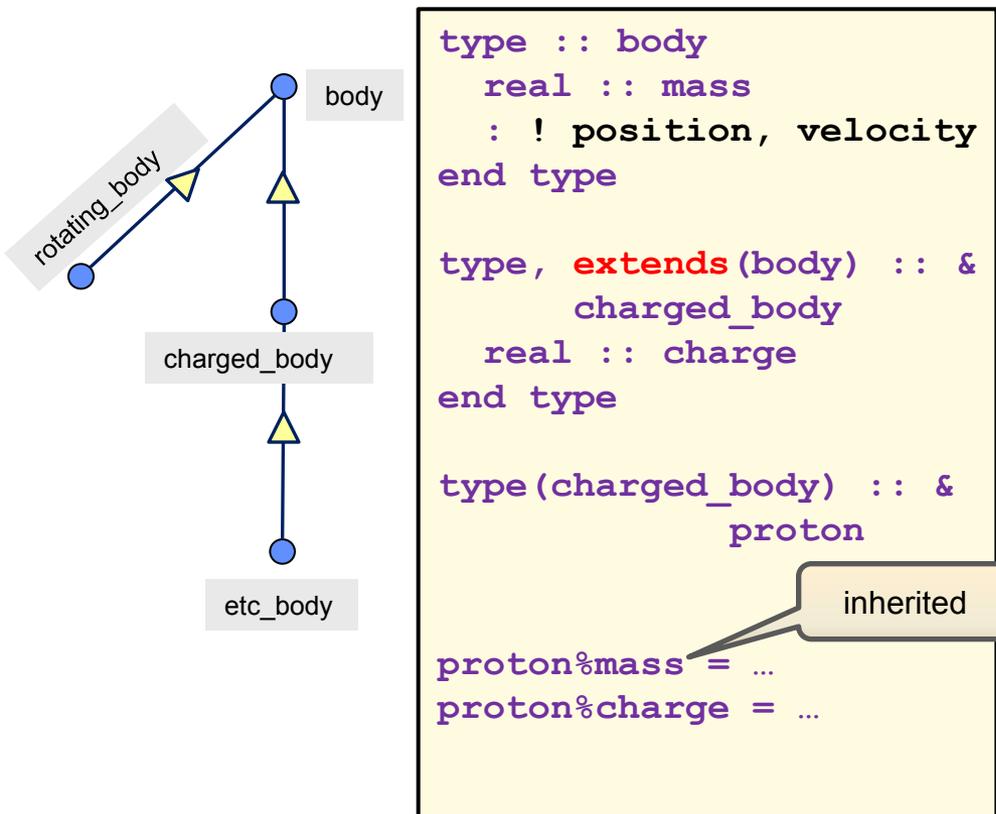
```
shared [1] Ctr o[THREADS];  
  
int main() {  
    if (MYTHREAD == p) {  
        o[MYTHREAD].d = (shared int *) \  
            upc_alloc(SZ*sizeof(int));  
    }  
    upc_barrier;  
    if (MYTHREAD == q) {  
        for (i=0; i<SZ; i++) {  
            o[p].d[i] = ... ;  
        }  
    }  
}
```

- local (per-thread) allocation into shared space
- program semantics the same as the CAF example on the previous slide

# Fortran Object Model (1)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

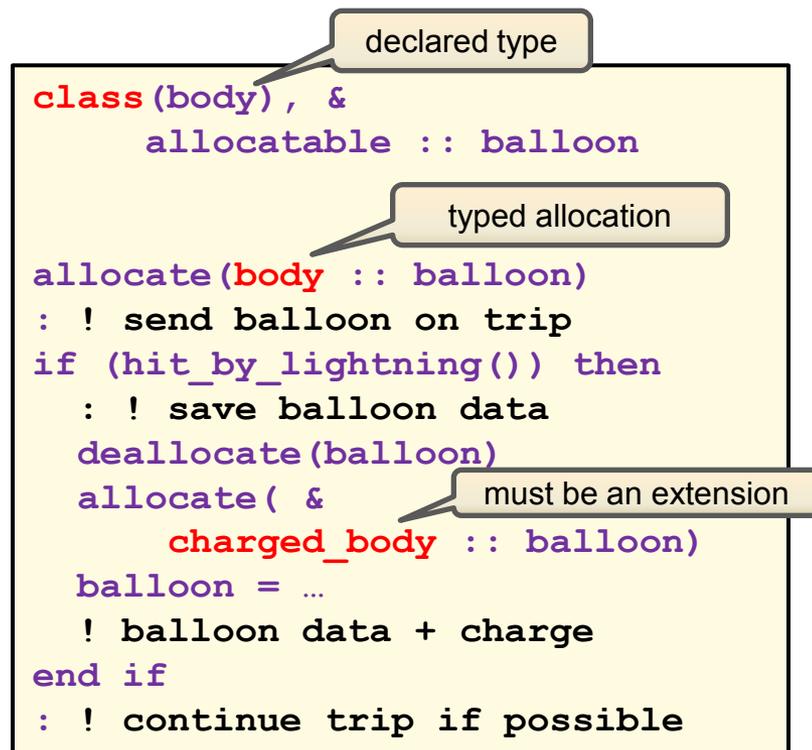
## • Type extension



- single inheritance (tree a DAG)

## • Polymorphic entities

- new kind of dynamic storage



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

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

# 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)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

## • Run time 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)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- **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

---

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Dynamic
- Advanced synchronization concepts
- Hybrid Programming

- **CAF**

- declaration of shared entity requires additional codimension (“fragmented data view”).
- Codimensions are very flexible (multi-dimensional).

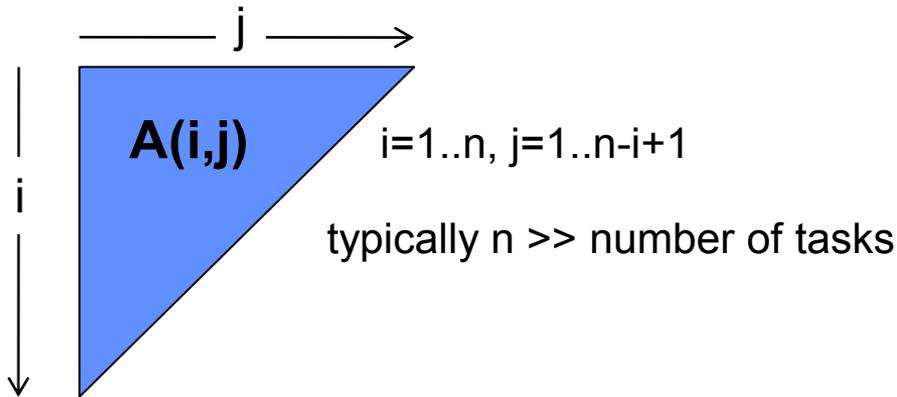
- **UPC**

- 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  $x[\dots][\text{THREADS}]$  implies round robin distribution
  - possibility of asymmetric distribution
- Multiple variants of dynamic allocation

# Second Exercise: Handling a triangular matrix (1)

- Basic PGAS concepts
- UPC and CAF basic syntax
  - Exercises
- Advanced synchronization concepts
- Hybrid Programming

- Consider a triangular matrix



- suggested data structure

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

Fortran

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

C

- Procedure:

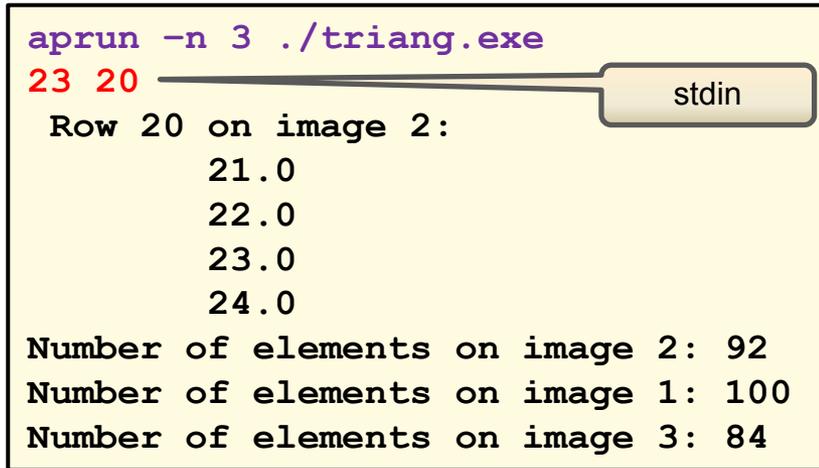
- make copy of `../skel/triangle_serial.f90` or `../skel/triangle_serial.c` to your working directory
- the program reads in matrix size and a row index, it then sets up  $A(i,j) = i+j$  and prints out the specified row
- parallelize this program in a manner that distributes data evenly across tasks
- 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:**
  - executed with three images

```
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
```

A terminal window with a yellow background. The first line is the command 'aprun -n 3 ./triang.exe' in purple. The second line is the input '23 20' in red. A grey callout box labeled 'stdin' has a line pointing to the input. The output shows 'Row 20 on image 2:' followed by four floating-point numbers: 21.0, 22.0, 23.0, and 24.0. Below that are three lines of element counts for images 2, 1, and 3.

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

---

# Advanced Synchronization Concepts

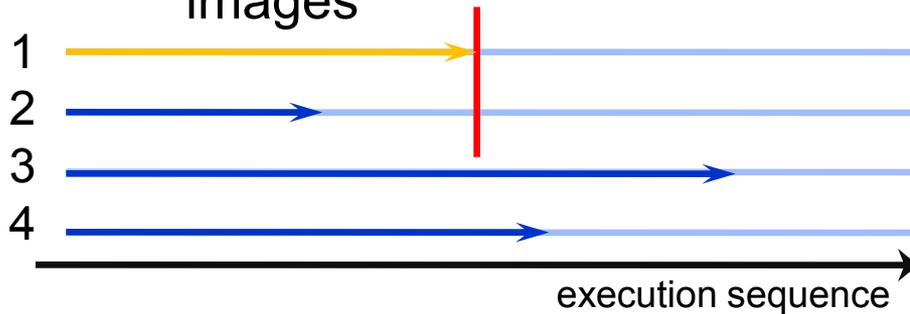
- o Partial synchronization
  - mutual exclusion
  - memory fences and atomic subroutines
  - split-phase barrier
- o Collective operations
- o Some parallel patterns and hints on library design:
  - parallelization concepts with and without halo cells
  - work sharing; distributed structures
  - procedure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

<https://fs.hlr.de/projects/rabenseifner/publ/SC2010-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

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

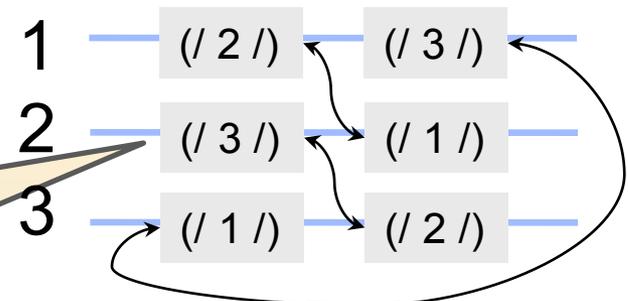
- UPC does not explicitly support this; note that in

```
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



# Example: Simple Master-Worker

- **Scenario:**

- one image sets up data for computations
- others do computations

- **Performance notes:**

- sending of data by image 1

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

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

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

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

# 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

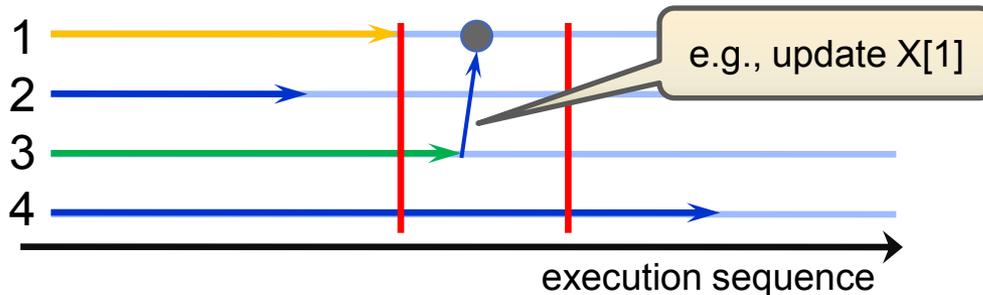
```
subroutine mysub(...)
:
  if (this_image() == 2) sync images (/ 1 /)
:
end subroutine
```

- 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:**

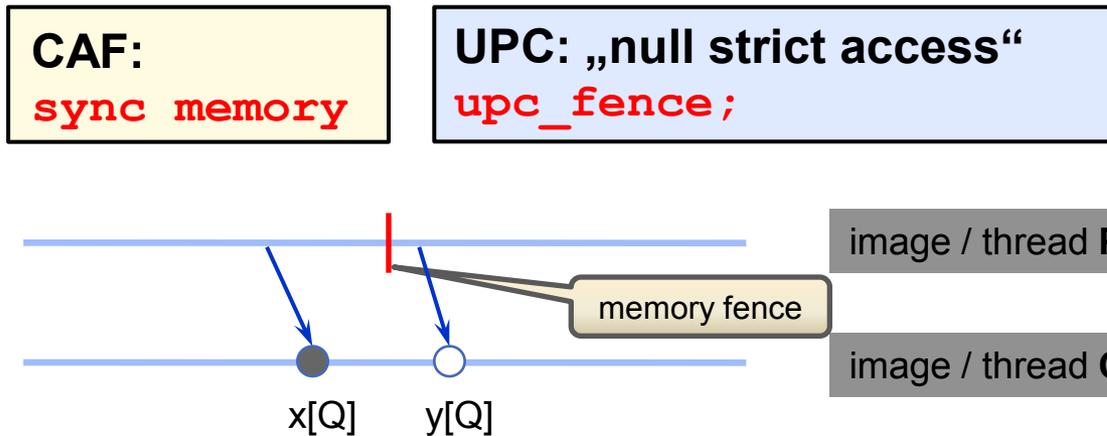
- segments corresponding to the code block ordered against one another
- this does **not** apply to preceding or subsequent code blocks
- may need additional synchronization to protect against race conditions

- **UPC:**

- use locks (see later)

# Memory fence

- **Target:** allow implementation of user-defined synchronization
- **Prerequisite:** subdivide a segment into two segments



**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

# Atomic subroutines and atomic types

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

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_R(ptr, value);
value = bupc_atomicI64_read_R(ptr);
```

- `shared int64_t *ptr;`
- `int64_t value;`
- unsigned and 32 bit integer types also available
- „\_R“ indicates relaxed memory model
- „\_S“ (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

# Example: Producer/Consumer

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

## • 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

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

## • 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_R(&ready, 1);
} else if (me == q) {
    val = 0;
    while (! val) {
        val = bupc_atomicI32_read_R(&ready);
    }
    upc_fence; ! B
    : // consume
}
```

## • further atomic functions:

- swap, compare-and-swap, fetch-and-add, fetch-and-*<logical-operation>*
- also suggested for CAF TR

# Recommendation

---

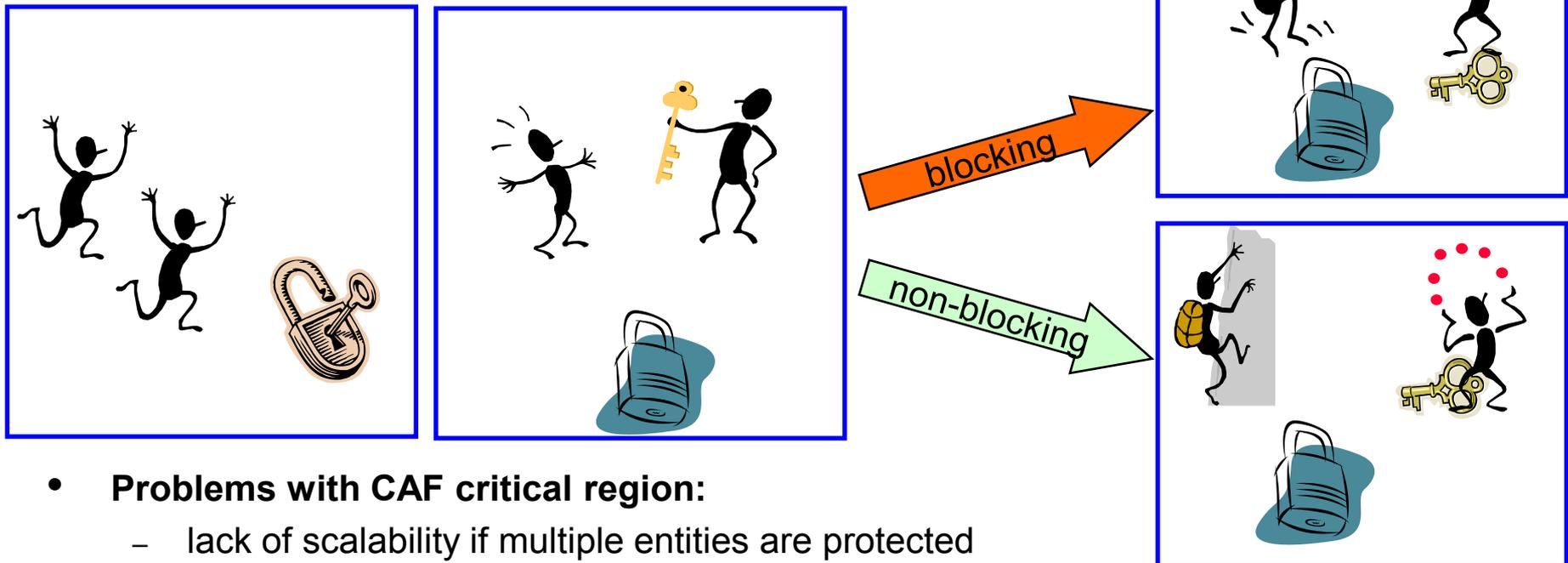
- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

- **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 mechanism for mutual exclusion

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

- **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

# Simplest examples for usage of locks

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

- **CAF:**

- coarray lock variable

```
use, intrinsic :: iso_fortran_env

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

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

do
  lock(lock[2], acquired_lock=got_it)
  if (got_it) exit
  : ! do stuff unrelated to any red balls
end do
: ! play with other red balls
unlock(lock[2])
```

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 a shared entity

lock = upc_all_lock_alloc();

upc_lock(lock);
: // play with red balls
upc_unlock(lock);

for (;;) {
  if (upc_lock_attempt(lock)) break;
  : // do other stuff
}
: // play with red balls
upc_unlock(lock);
upc_lock_free(lock);
```

collective call same result on each thread

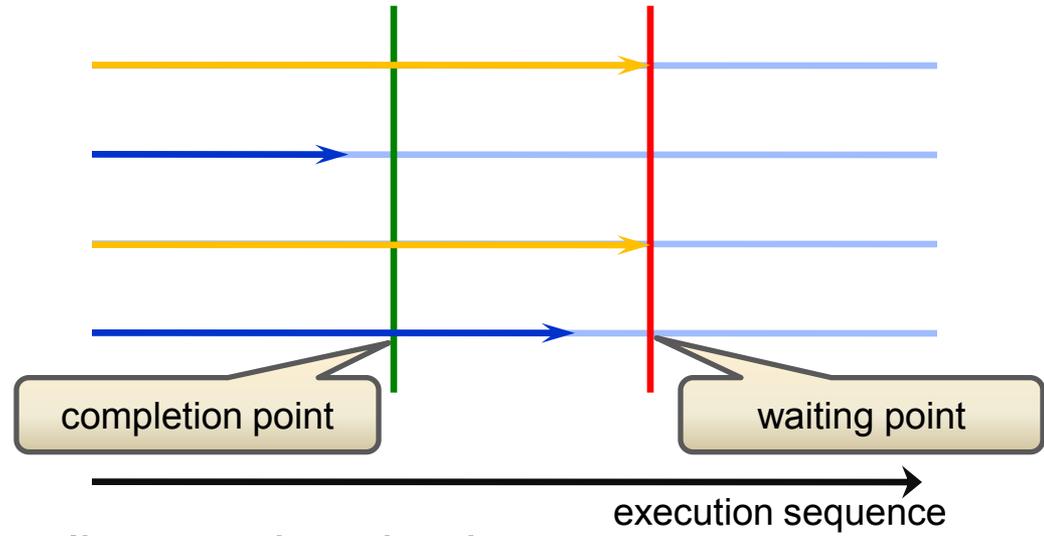
- thread-individual lock generation is also possible (non-collective)
- lock/unlock imply memory fence

# UPC: Split-phase barrier

- **Separate barrier completion point from waiting point**
  - this allows threads to continue computations once all others have reached the completion point → ameliorate 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` implies synchronization
- collective – **all** threads must execute sequence
- **CAF:**
  - presently does not have this facility in statement form
  - could implement using locks and event counts

# UPC: Memory consistency modes

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming and outlook

## • 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)

```
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

- code section level:

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

- program level

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

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

# What strict memory consistency does and doesn't do for you

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

- „strict“ **cannot** prevent all race conditions

- example: „ABA“ race

```
strict shared int flag;  
int val;
```

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 assure that changes on (complex) objects

- appear to be atomic to other threads
- appear in the same order on other threads

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

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

may obtain (0, 1), (0, 2)  
or (1, 2), but **not** (2, 1)

---

# Advanced Synchronization Concepts

- Partial synchronization
  - mutual exclusion
  - memory fences and atomic subroutines
  - ~~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.hlr.de/projects/rabenseifner/publ/SC2010-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`

```
UPC - { IN } - { NOSYNC
      { OUT } - { MYSYNC
                ALLSYNC
```

- **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:**

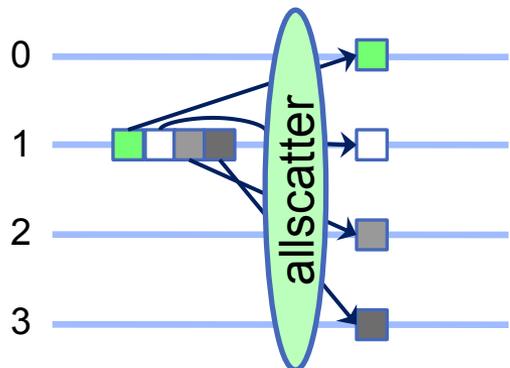
- `UPC_IN_NOSYNC | UPC_OUT_MYSYNC`
- `UPC_IN_NOSYNC` same as `UPC_IN_NOSYNC | UPC_OUT_ALLSYNC`
- `0` same as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`

# Collectives (2): Example for redistribution

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

## • 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]
```

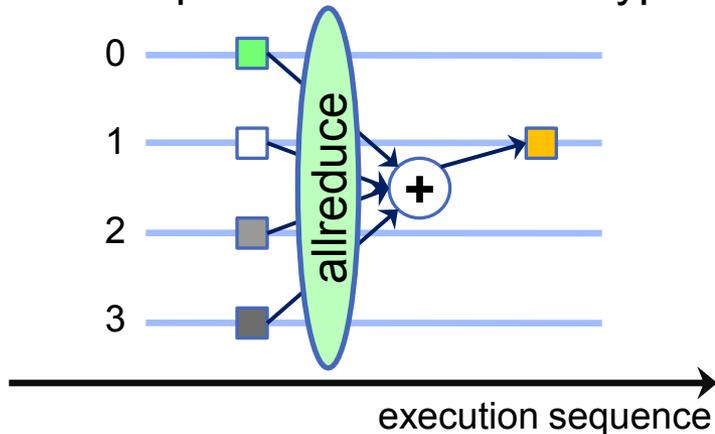
simpler, but no asynchronous execution possible

# Collectives (3): Reductions

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

## • Reduction concept:

- distributed set of objects
- operation defined on type



- calculate destination object in shared space

## • Availability:

- UPC only
- CAF: need to roll your own for now (future CAF may include this feature)

## • Reduction types

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

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

```
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  
if > 0

- `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

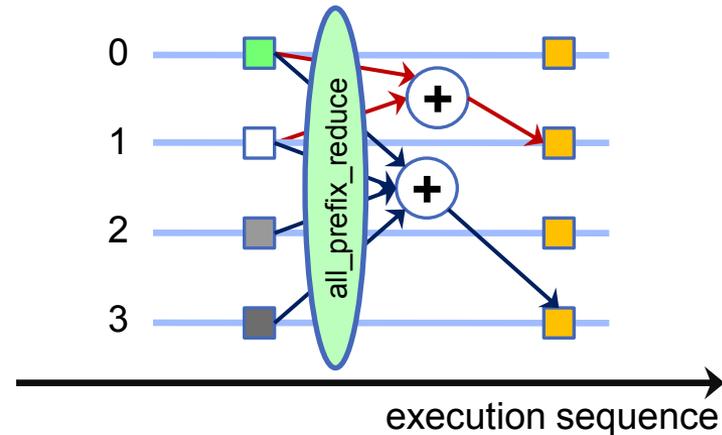
- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
- Hybrid Programming

## • Redistribution functions

- `upc_all_broadcast()`
- `upc_all_gather_all()`
- `upc_all_gather()`
- `upc_all_exchange()`
- `upc_all_permute()`

## • Prefix reductions

- `upc_all_prefix_reduce`**T**()
- semantics:



→ consult the UPC language specification for details

for `UPC_ADD`,  
thread  $i$  gets  
(thread-dependent result)

$$\sum_{k=0}^i d_k$$

---

# Advanced Synchronization Concepts

- Partial synchronization
  - mutual exclusion
  - memory fences and atomic subroutines
  - 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

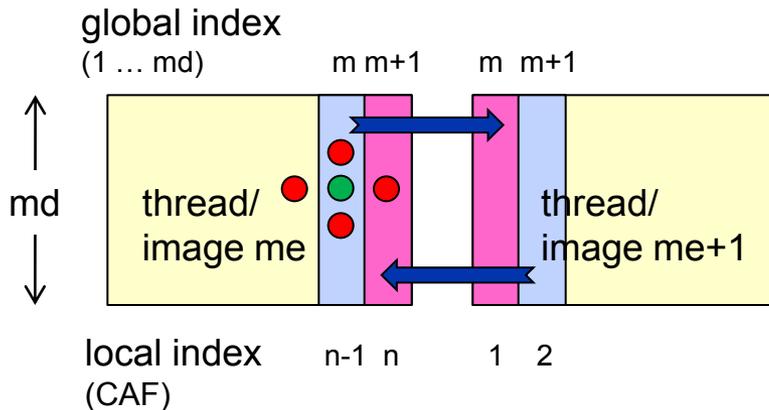
<https://fs.hlr.de/projects/rabenseifner/publ/SC2010-PGAS.html>

# Work sharing (3) data exchange

- Basic PGAS concepts
- UPC and CAF basic syntax
- Parallel Patterns and Practices
- Hybrid Programming

## • Halo data

- context: stencil evaluation
- example: Laplacian



(halo size is 1)

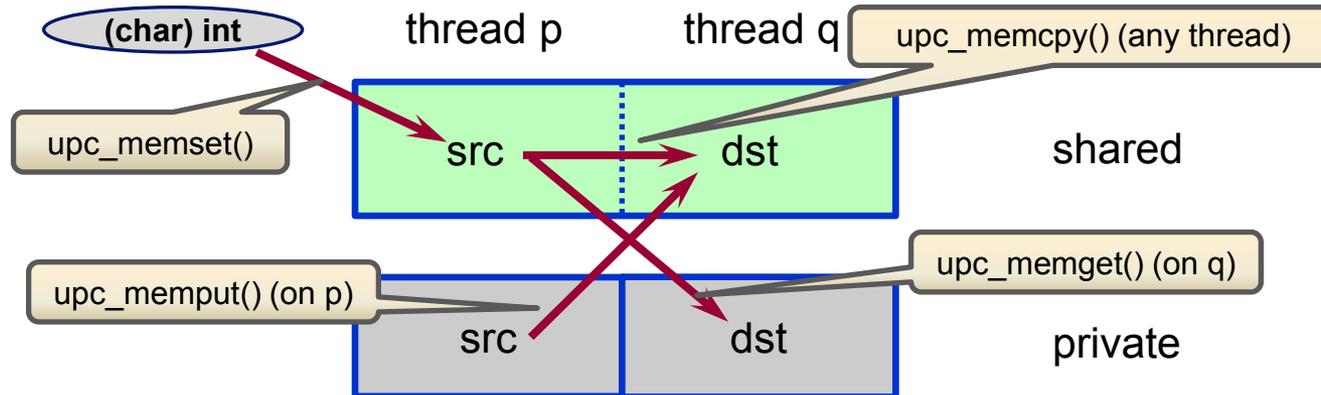
- data exchange (blue arrows) required e.g. for iterative updates

## • CAF halo update

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

- uses „pull“ style
- what about „push“?
- 1-d data distribution: not the most efficient way

# 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_mempup(shared void *dst,
               void *src, size_t n);
void upc_memset(shared void *dst,
               int c, size_t n);
```

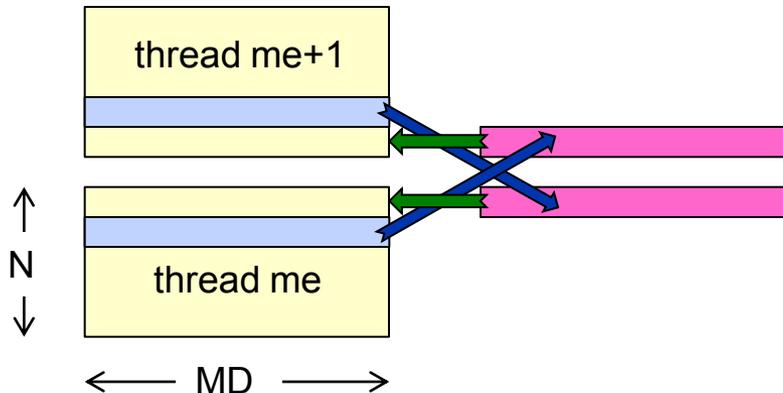
# Work sharing (4)

## data exchange avoiding haloed shared data

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Parallel Patterns and Practices**
- Hybrid Programming

### • Use following data layout

```
double a[N][MD];
shared [*] double
    a_top[THREADS][MD],
    a_bot[THREADS][MD];
```



- does not require entire data field to be shared

**Note:**

for 2-D blocking this is not fun. A strided block transfer facility would be welcome.

### • Communication code

```
if (MYTHREAD+1 < THREADS) {
    upc_memput(&a_bot[MYTHREAD+1][0],
              &a[N-2][0], MD);
}
if (MYTHREAD-1 > 0) {
    upc_memput(&a_top[MYTHREAD-1][0],
              &a[1][0], MD);
}
upc_barrier;
if (MYTHREAD > 0) {
    upc_memget(&a[0][0],
              &a_bot[MYTHREAD][0], MD);
}
if (MYTHREAD < THREADS) {
    upc_memget(&a[N-1][0],
              &a_top[MYTHREAD][0], MD);
}
```

stencil calculation epoch ends

- maintains one-sided semantics, but one more copy needed

# 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;
  for (i=0; i<n; i++) {
    ... = w[i] + ...
  }
  // exchange data
  upc_barrier;
  // etc.
}
```

- assume 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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Parallel Patterns and Practices
- Hybrid Programming

## • 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 discontinuous

## • UPC

```
shared [*] float x[THREADS][NDIM]

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

cast → loss of phase information

# Factory procedures

- **CAF:**  
allocatable dummy argument

```
subroutine factory(wk, ...)
  real, allocatable :: wk(:)[: ]
  : ! determine size
  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
  wk = (shared float *)
    upc_all_alloc(THREADS,
    sizeof(float)*n);
  : // fill wk with data
  return wk;
}
```

if block size  
>1 supported

- 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

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Parallel Patterns and Practices**
- Hybrid Programming

- **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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Parallel Patterns and Practices
- Hybrid Programming

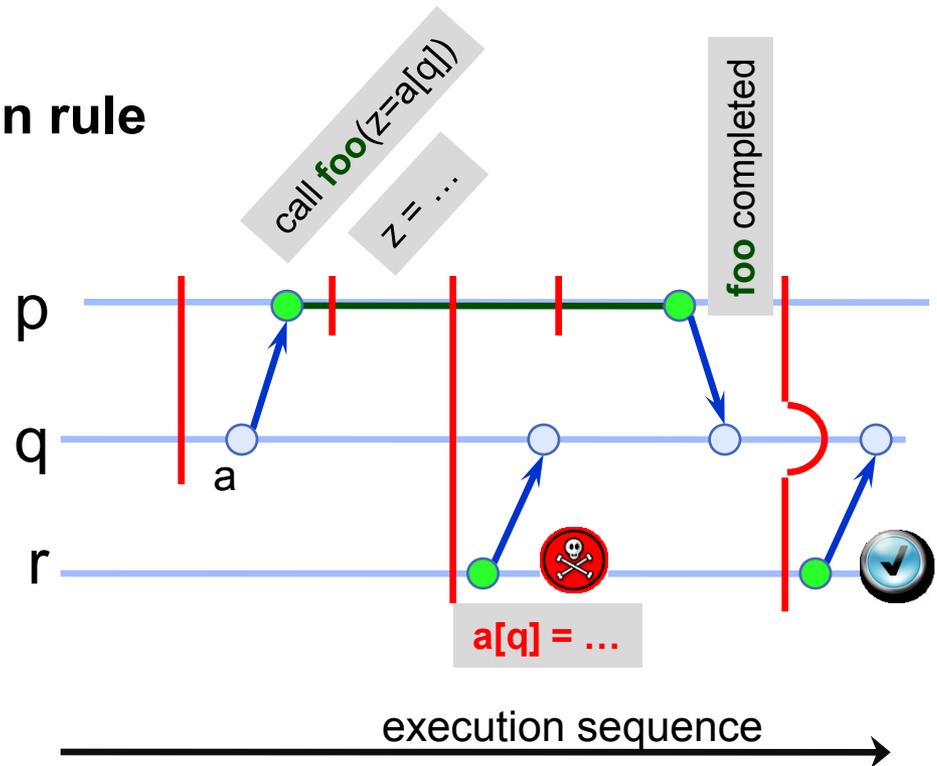
- **Assumptions:**

- 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)

- Basic PGAS concepts
- UPC and CAF basic syntax
- Parallel Patterns and Practices
- Hybrid Programming

- Irregular data distribution

- use a data structure
- recursive processing

- UPC example:

- binary tree

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

regular „serial“ type definition

- prerequisite: ordering relation

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

- Constructor for Tree object

- must be called by **one** thread **only**

```
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));  
    return this;  
}
```

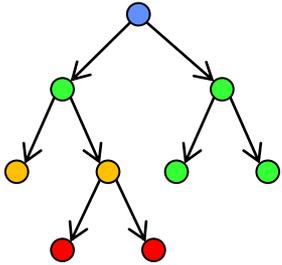
- initialize storage for lock and data components

# Distributed Structures (2)

- Basic PGAS concepts
- UPC and CAF basic syntax
- Parallel Patterns and Practices
- Hybrid Programming

- **Concurrent population**

- locking ensures race-free processing



color ↔ thread number

copy object to  
(remote) shared entity

```
void insert(shared Tree *this, Content *stuff) {
    upc_lock(this->lk);
    if (this->left) {
        upc_unlock(this->lk);
        if (lessthan((Content *) this->data, stuff)) {
            insert(this->left, stuff);
        } else {
            insert(this->right, stuff);
        }
    } else {
        this->left = Tree_init();
        this->right = Tree_init();
        upc_mempup(this->data, stuff, sizeof(Content));
        upc_unlock(this->lk);
    }
}
```

invoke  
constructor

# Distributed Structures (3)

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Parallel Patterns and Practices**
- Hybrid Programming

- **Assumption**

- structure is written once or rarely
- many **operations** performed on entries, in access epochs separated from insertions

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

guarantee  
locality

- traversal must be executed by all threads which called the constructor to be complete

- **CAF:**

- cannot easily implement this concept with coarrays
- shared objects on one image only not supported
- klugey workaround using pointer components of coarrays is possible

# Third exercise: Manual reduction and prefix reduction

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
  - Exercises
- Hybrid Programming

- 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?

# Fourth Exercise:

## Heat conduction in 2 dimensions

---

- Basic PGAS concepts
- UPC and CAF basic syntax
- **Advanced synchronization concepts**
  - Exercises
- Hybrid Programming

- **Make a copy of serial programs into your working directory**
  - `cp ../skel/heat_serial.c heat_upc.c`
  - `cp ../skel/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

# Real Applications and Hybrid Programming

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

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

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

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

# The NPBs in UPC are useful for studying various PGAS issues

---

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

- **Using customized communication to avoid hot-spots**
  - UPC Collectives do not support certain type of 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/SeaStar2** interconnect, but almost no difference is observable on **Sun Constellation Cluster/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

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

- **UPC Collectives might not support certain type 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

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

- **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)`
  - `void bupc_waitsync(upc_handle_t handle);` - wait for completion
  - Asynchronous version 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

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

- Using strided communication is generally an improvement
- Message aggregation reduces the number of messages, but introduces the packing/unpacking overhead
- Message aggregation increases programming effort.
- Example:

## Fine-grain communication Thread A → Thread B

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

## Message Aggregation Thread A:

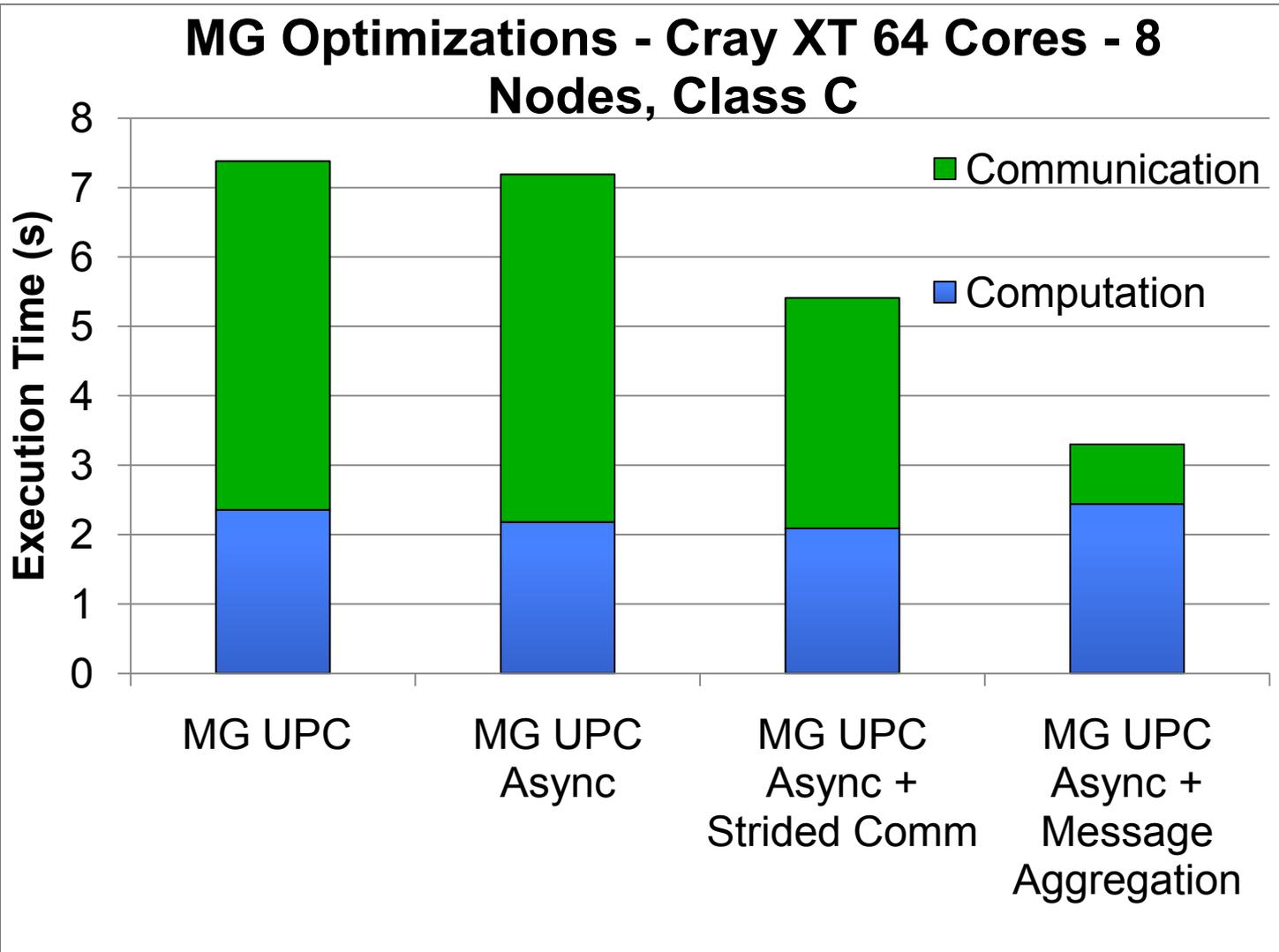
```
buff = pack(u);  
upc_mempush( &k(0),  
            &buff,  
            n1*n2*sizeof(double));  
upc_barrier;
```

## Thread B:

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

# MG message aggregation is significantly better on Cray SeaStar2 interconnect

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



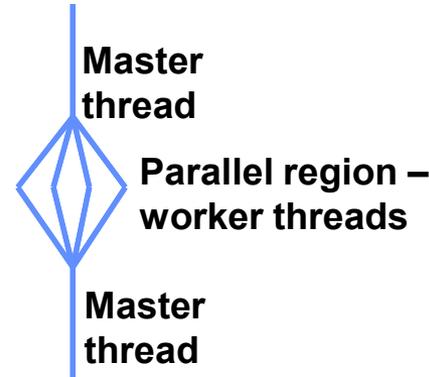
- MG message aggregation had almost no difference on Ranger InfiniBand interconnect

# UPC – Shared Memory Programming reduces communication time

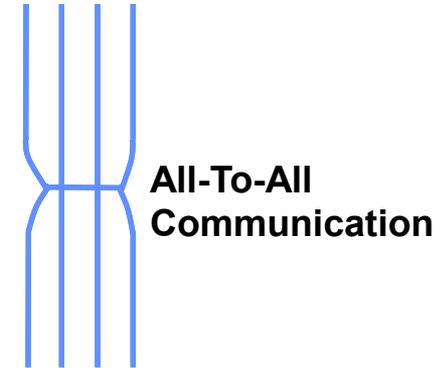
- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization concepts
- Hybrid Programming

- UPC initially designed for distributed systems
- UPC capable of exploiting shared memory (OMP-like) programming style & avoiding explicit communication

OMP – Shared Memory style



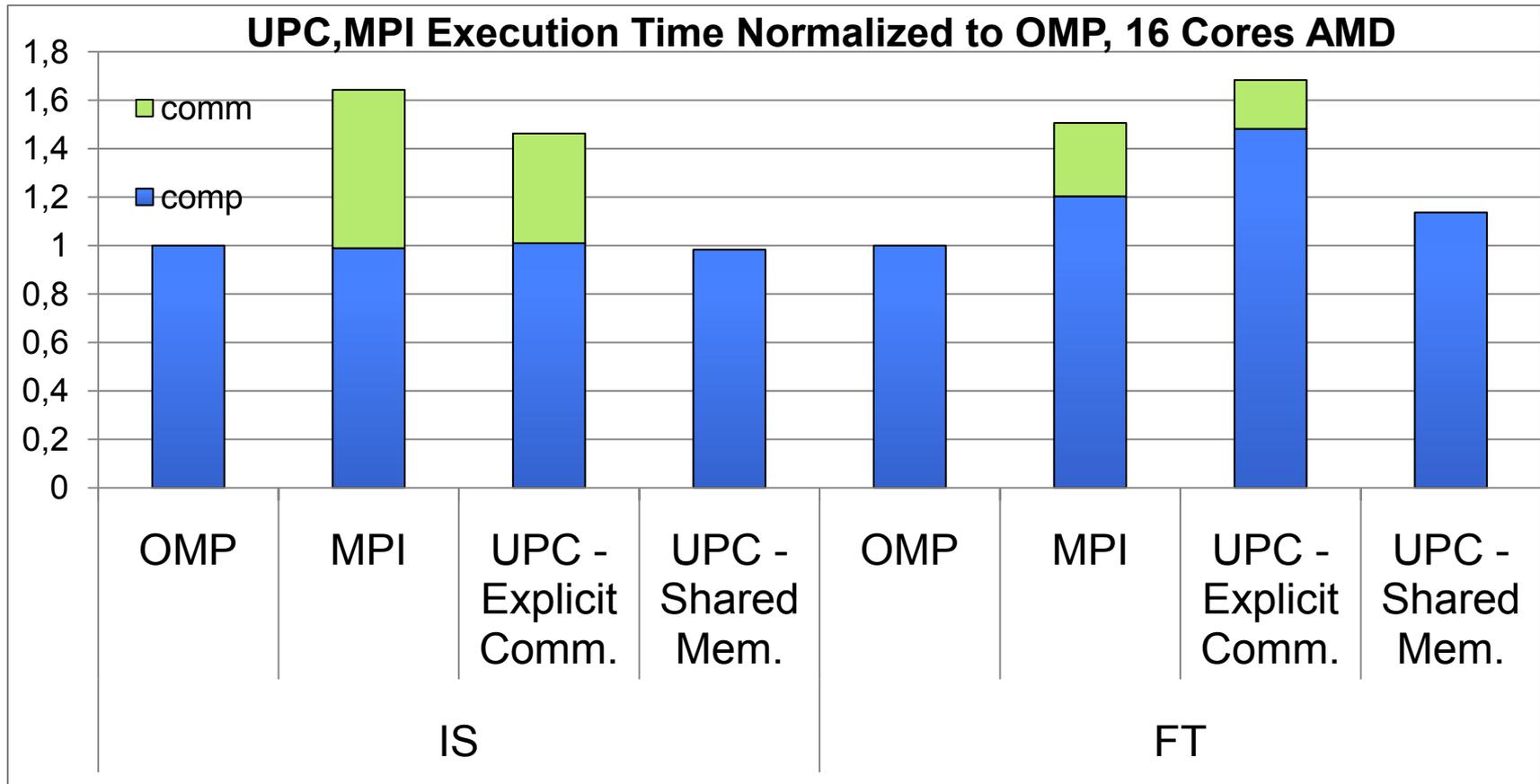
MPI – Explicit Communication



- **Drawback: reduced memory utilization**
  - UPC threads are required to equally participate in shared-heap allocation
  - In the UPC-shared memory model, only part of the heap allocated by the master thread is used, resulting in memory underutilization
  - 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
- Hybrid Programming



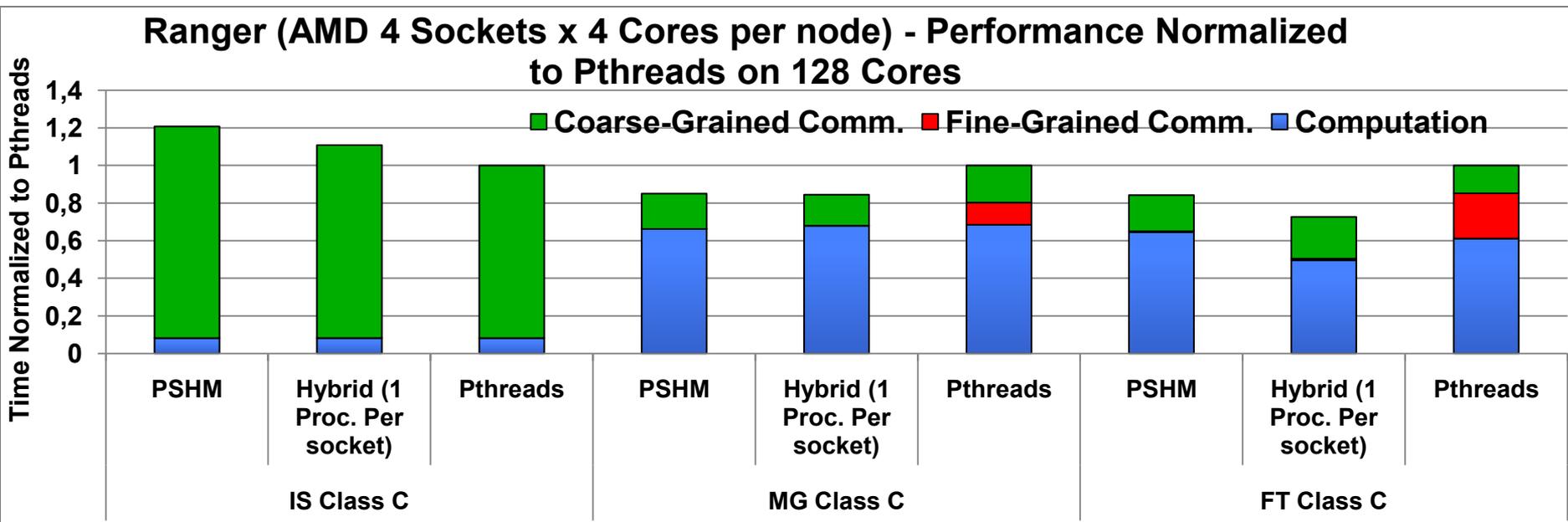
# BUPC language-level threads can be mapped to Pthreads and/or Processes

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

- **Pthreads** – shared memory communication through shared address space
- **Processes** – shared memory communication through shared memory segments (POSIX, SYSV or mmap) called PSHM
- **NPBs performance depend on Pthreads/Processes**
  - Pthreads share one network endpoint; PSHM - one network endpoint per process
  - Due to sharing of one network endpoint, pthreads experience message throttling
  - Processes (PSHM) can inject larger number of messages into the network (large number of messages can sometimes decrease overall bandwidth)
  - PSHM avoids contention overhead when interacting with external libraries/drivers

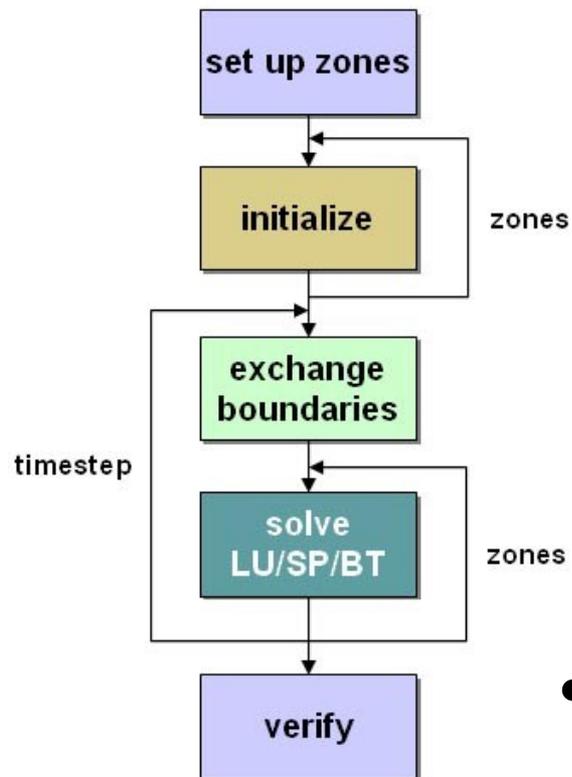
# Mix of processes and pthreads is often required for achieving the best performance

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



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](http://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

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

- **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

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

- 
- MPI is designed to allow coexistence with other parallel programming paradigms and uses the same SPMD model as CAF:
    - ➔ MPI and 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

---

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

- **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

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

```
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
```

main.F90

```
subroutine mpi_routine1...
subroutine mpi_routine2 ...
```

mpi.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
```

caf.F90

```
# 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

---

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

- **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.

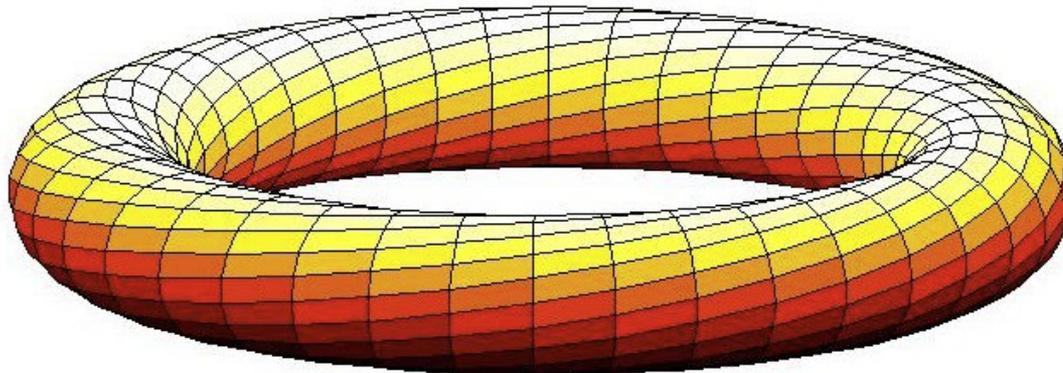
Dot product coding from “Hybrid Parallel Programming with MPI and Unified Parallel C” by James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur

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

---

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

- 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)

---

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

- 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

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

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.hirs.de/projects/rabenseifner/publ/SC2010-PGAS.html>

# README – UPC on Cray XT...: UPC / PGI

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Presenters
  - Abstract
  - Literature

Initialization: module load `bupc`

Interactive PBS shell:

In general:

```
qsub -I -q debug -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

In the SC tutorial

```
qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

Again to the working directory:

```
cd $PBS_O_WORKDIR
```

Compilation:

```
upcc -O -pthreads=4 -o myprog myprog.c
```

Parallel Execution:

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

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

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

Exercise 1

Exercise 2

Exercise 3/4

If your batch job was initiated

with 8 cores, i.e., with: `-lmppwidth=8,mppnppn=4`

```
upcrun -n 8 -cpus-per-node 4 ./myprog
```

# README – UPC on Cray XT...: Cray UPC

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Presenters
  - Abstract
  - Literature

Initialization: module switch PrgEnv-pgi PrgEnv-cray

Interactive PBS shell:

In general:

```
qsub -I -q debug -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

In the SC tutorial

```
qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

Again to the working directory:

```
cd $PBS_O_WORKDIR
```

Compilation:

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

Parallel Execution:

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

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

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

If your batch job was initiated

with 8 cores, i.e., with: -lmppwidth=8,mppnppn=4

```
aprun -n 8 -N 4 ./myprog
```

Exercise 1

Exercise 2

Exercise 3/4

# README – UPC on Cray XT...: Cray Fortran

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Presenters
  - Abstract
  - Literature

Initialization: module switch PrgEnv-pgi PrgEnv-cray

Interactive PBS shell:

In general:

```
qsub -I -q debug -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

In the SC tutorial

```
qsub -I -q special -lmppwidth=4,mppnppn=4,walltime=00:30:00 -V
```

Again to the working directory:

```
cd $PBS_O_WORKDIR
```

Compilation:

```
crayftn -hcaf -o myprog myprog.f90
```

Parallel Execution:

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

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

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

If your batch job was initiated

with 8 cores, i.e., with: -lmppwidth=8,mppnppn=4

```
aprun -n 8 -N 4 ./myprog
```

Exercise 1

Exercise 2

Exercise 3/4

# hello\_upc.c and hello\_caf.f90

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- **Appendix**
  - Exercises
  - Presenters
  - Abstract
  - Literature

```
#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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Presenters
  - Abstract
  - Literature

- Matrix object declaration and initialization code

```
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

- ../solutions/triangular.f90 (Fortran)
- ../solutions/triangular.upc (UPC)

Exercise 2

# Manual reduction: mod\_reduction\_simple.f90

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Presenters
  - Abstract
  - Literature

- 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
```

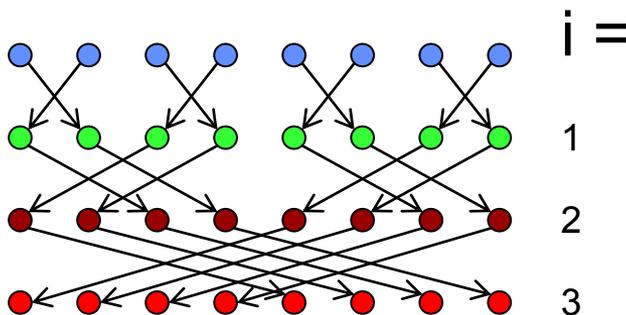
# Manual reduction (2)

- **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:**

- ../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 = shiffl(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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- **Appendix**
  - Exercises
  - Presenters
  - **Abstract**
  - Literature

**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 hybrid programming 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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Abstract
  - Presenters
  - Literature

- **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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Abstract
  - Presenters
  - Literature

- **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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Abstract
  - Presenters
  - Literature

- **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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- Appendix
  - Exercises
  - Abstract
  - Presenters
  - Literature

- **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 2001-2010 (5 days) - G. Hager, R. Bader et al: Parallel Programming and Optimization on High Performance Systems
- LRZ Munich (2009) (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

- Basic PGAS concepts
- UPC and CAF basic syntax
- Advanced synchronization
- Hybrid Programming
- **Appendix**
  - Exercises
  - **Presenters**
  - Abstract
  - Literature

- **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](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>