CRAY

PGAS Programming on Cray XK6

Roberto Ansaloni
Cray Italy
roberto@cray.com

Agenda: Day3 - May 25th

09:00-10:00	CAF and UPC introduction
10:00-10:30	Cray Programming Environment and PGAS compilers
10:30-11:00	Coffee break
11:00-11:30	Cray performance tools for MPI and PGAS code development and tuning
11:30-12:30	Users talks & discussion (Romain Teyssier, Will Sawyer)
12:30-13:30	Lunch break
13:30-15:00	PGAS lab and wrap up

Acknowledgments

- This work relies on material developed by some colleagues at Cray, in particular by
  - Luiz DeRose
  - Alistair Hart
  - Bill Long
  - Heidi Poxon
  - Harvey Richardson
  - Rick Slick
  - Nathan Wichmann

Fortran Coarray and UPC Introduction

Roberto Ansaloni

CRAY

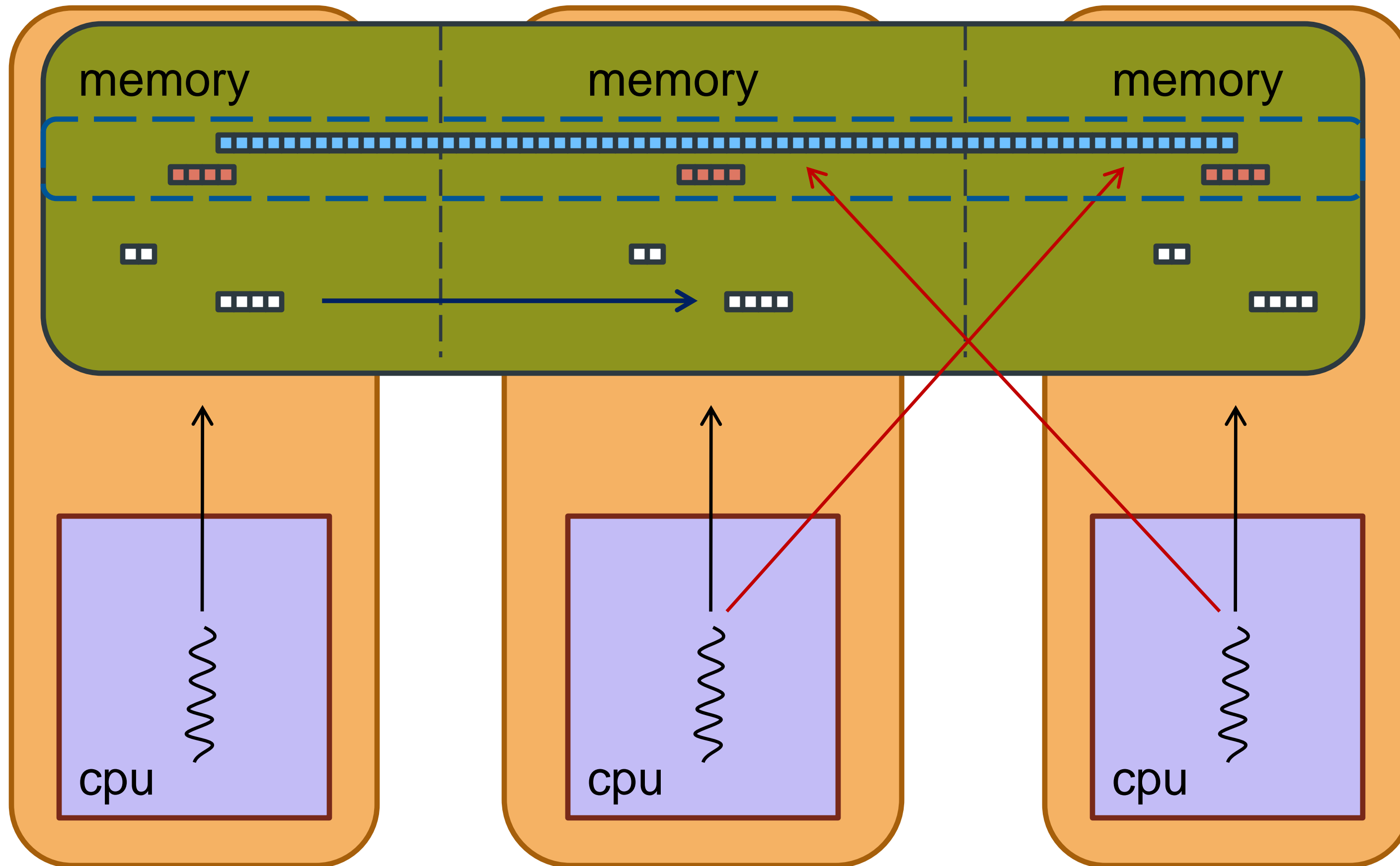
Partitioned Global Address Space Languages

- **Explicitly-parallel programming model with SPMD parallelism**
  - Fixed at program start-up, typically 1 thread per processor
- **Global address space model of memory**
  - Allows programmer to directly represent distributed data structures
- **Address space is logically partitioned**
  - Local vs. remote memory (two-level hierarchy)
- **Programmer control over performance critical decisions**
  - Performance transparency and tunability are goals

Partitioned Global Address Space Model

- Access to remote memory is a full feature of the PGAS language
  - Type checking
  - Opportunity to optimize communication
- Participating processes/threads have access to local memory via standard program mechanisms
  - No performance penalty for local memory access
- Single-sided programming model more natural for some algorithms
  - and a good match for modern networks with RDMA

PGAS




```
A[2] = A
```

PGAS Languages

- **Fortran 2008**
  - Now incorporates coarrays
  - New codimension attribute for objects
  - New mechanism for remote access:
    `a(:)=b(:)[image] ! Get b from remote image`
  - Replication of arrays on every image with "easy and obvious" ways to access those remote locations.
- **UPC**
  - Specification that extends the ISO/IEC 9899 standard for C
  - Participating "*threads*"
  - New *shared* data structures
  - Language constructs to divide up work on shared data
  - Philosophically different from Fortran coarrays
    - Compiler intimately involved in detecting and executing remote references
  - Flexible, but filled with challenges like pointers, a lack of true multidimensional arrays, and many options for distributing data

Fortran coarrays

Coarrays in Fortran

- Coarrays were designed to answer the question:

  What is the smallest change required to convert Fortran into a robust and efficient parallel language?

- The answer:

  A simple syntactic extension: [ ]

- It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules

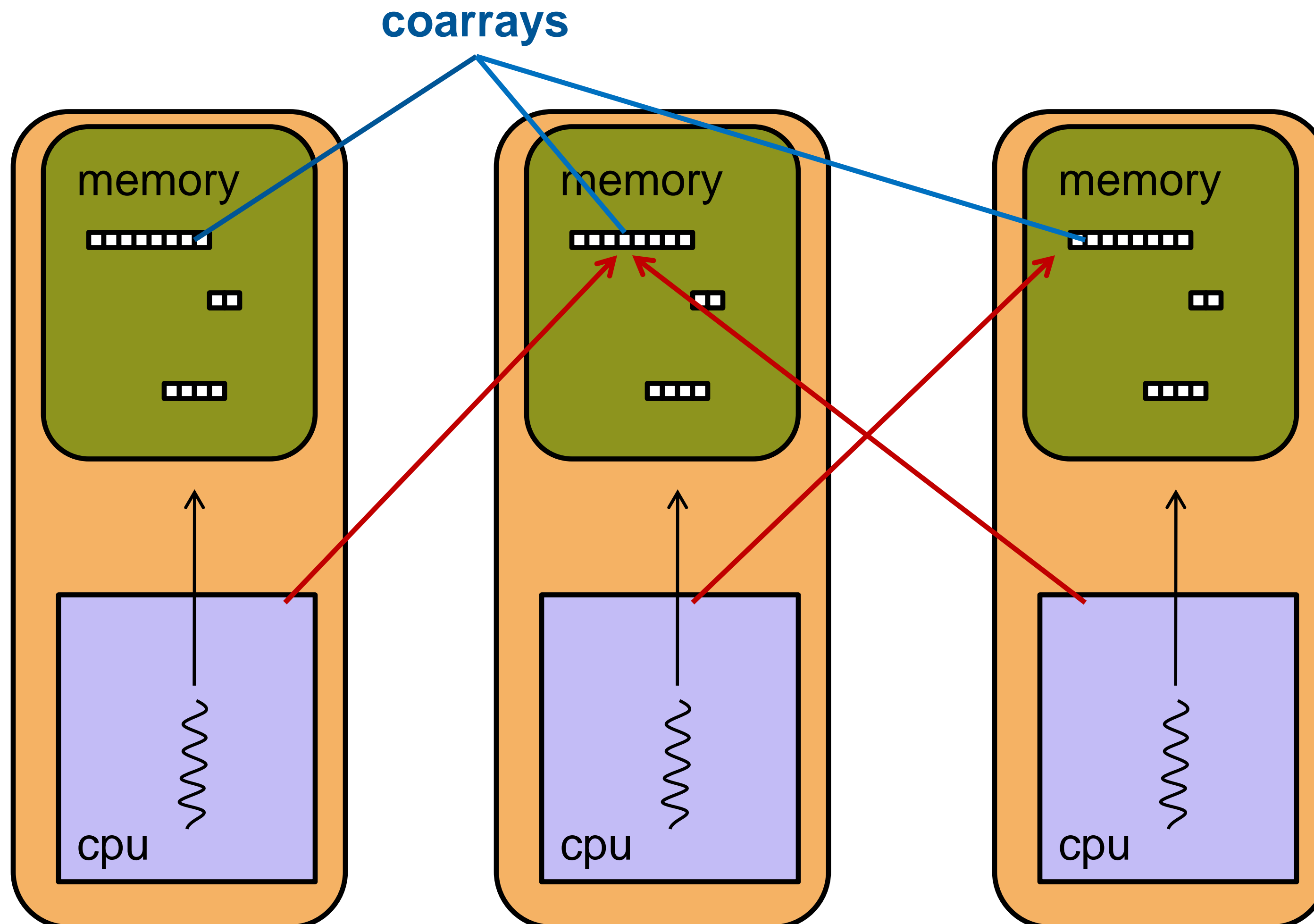
Coarrays background

- **Based on early work by Bob Numrich on the Cray T3D**
  - The Cray T3D address space and how to use it (1994)
- **Originally there were get/put functions within a library**
  - Evolved into what is known today as **SHMEM** library
- **Initial proposal of a Fortran extension: F--**
  - F− − : A parallel extension to Cray Fortran.
    Scientific Programming 6, 275-284. (1997)
- **Introduced in current form by Numrich and Reid in 1998 as a simple extension to Fortran 95 for parallel processing**
- **Now part of the Fortran standard: ISO/IEC 1539-1:2010**
  - Additional features are expected to be published in a Technical Specification in due course. (collectives)
  - Various vendor implementations (Intel) and Open Source projects (g95, gfortran) underway
  - Available on Cray compilers since its introduction

Basic execution model and features

- Program executes as if replicated to multiple copies with each copy executing asynchronously (SPMD)
- Each copy (called an **image**) executes as a normal Fortran application
- New object indexing with [ ] can be used to access objects on other images.
- New features to inquire about image index, number of images and to synchronize

Coarray execution model

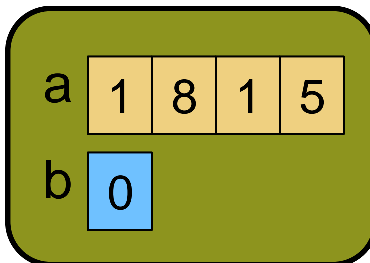


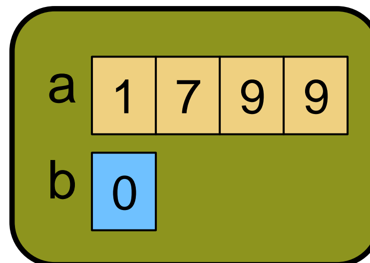
Remote access with square bracket indexing: a(:)[2]

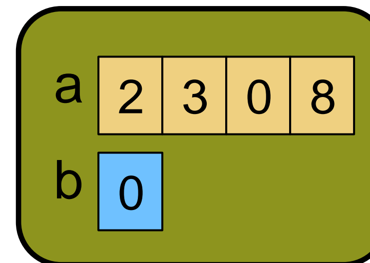
Basic coarrays declaration and usage

- Codimensions are used to indicate data allocated on specific processors (**images**)

```
integer :: b
integer :: a(4)[*] !coarray
```





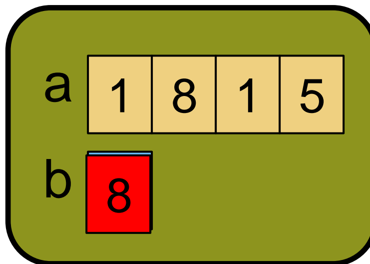


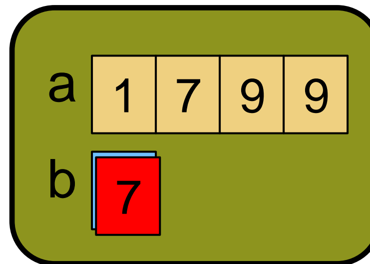
- Coarrays need to have the same dimension on all images

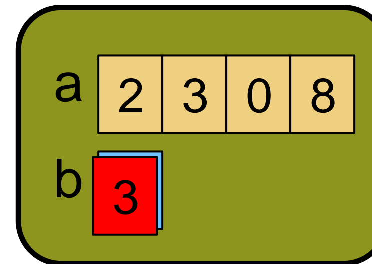
Basic coarrays declaration and usage

- Local reference

```
b = a(2)
```





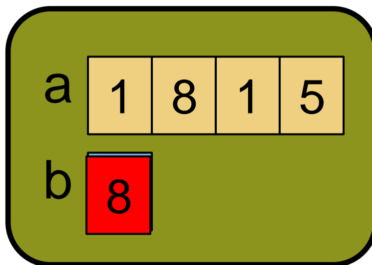


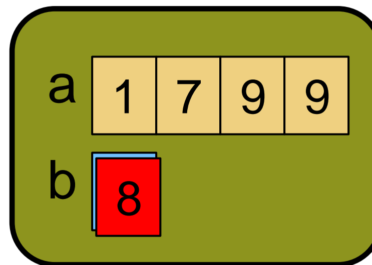
- References without codimensions [ ] are local
- b is set to second element of a on each image

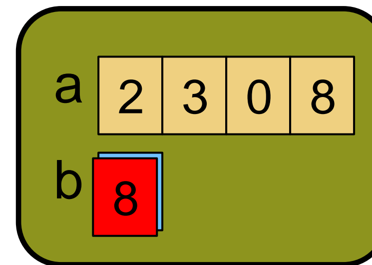
Basic coarrays declaration and usage

- Some data movements

```
b = a(4)[3]
```







- [ ] indicates access to remote coarray data
- Each b is set to fourth element of array a on image 3

Coarray declarations

- Codimensions can be added to any type of valid Fortran variables

```
real :: residual[*]                 ! Scalar coarray

real, dimension(100), codimension[*] :: x,y

type (color) map(512,512)[*]

character(len=80), allocatable ::search_space(:)[:]
allocate( search_space(2000)[*] )

real, allocatable :: a(:,:)[:,:]
allocate( a(1000,2000)[px,*] )
```

Image execution

- **Functions are provided to return number of images and index of executing image**
- **num_images()**
  - Returns the number of images (processing elements)
  - num_images() = number of MPI ranks
- **this_image()**
  - Returns the number of the current image
  - 1 ≤ this_image() ≤ num_images()
  - this_image = MPI rank + 1
- **Allow images to organize problem distribution and to operate independently**

Example: read and distribute and array from file

- Read n elements at a time and distribute

```
double precision, dimension(n) :: a
double precision, dimension(n) :: temp[*]

if (this_image() == 1) then
  do i=1, num_images()
    read *,a
    temp(:)[i] = a
  end do
end if
!! Is this safe ???
temp = temp + this_image()
```

Example: read and distribute and array from file

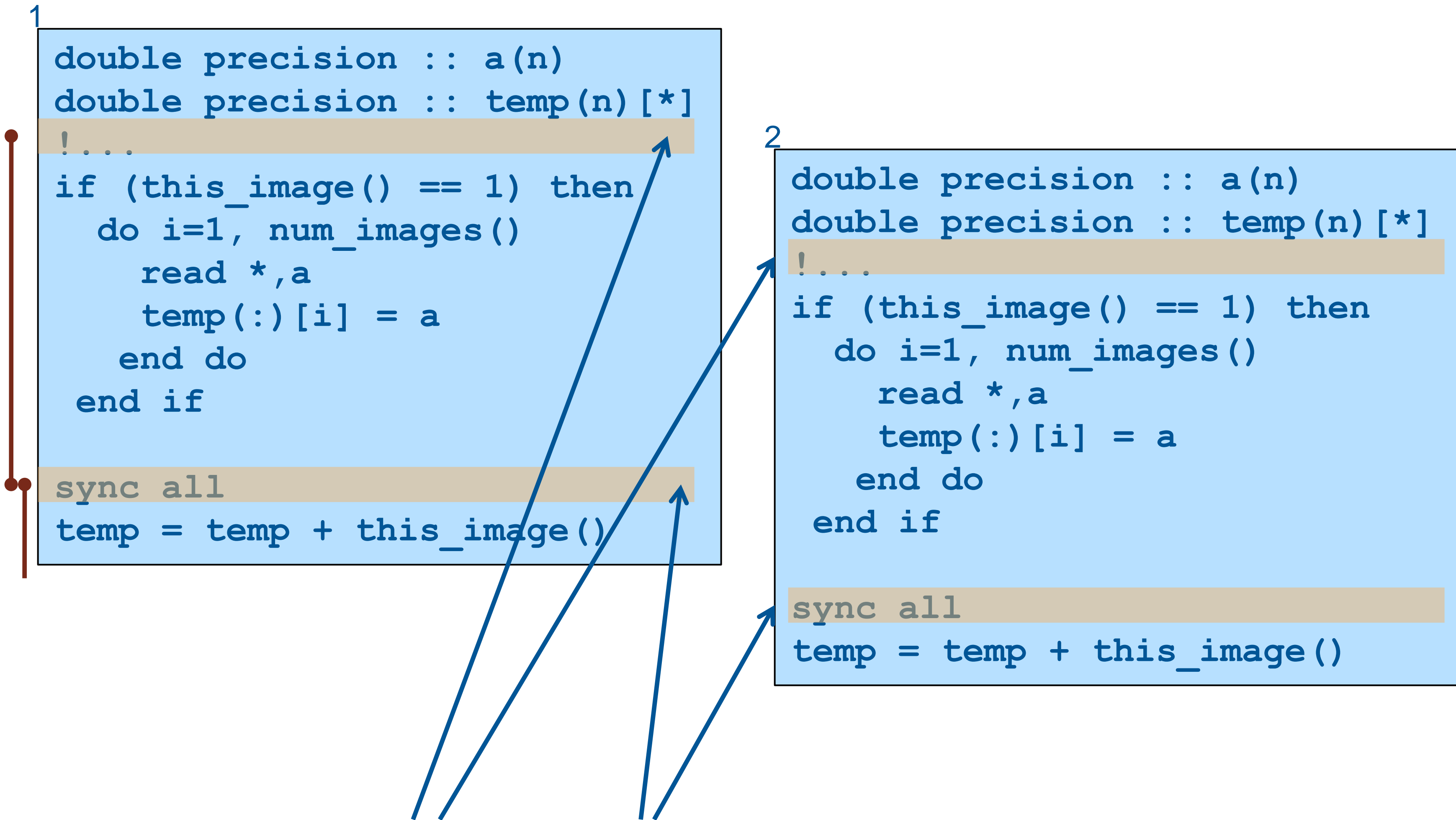
- Need to sync between images before using temp

```
double precision, dimension(n) :: a
double precision, dimension(n) :: temp[*]

if (this_image() == 1) then
   do i=1, num_images()
      read *,a
      temp(:)[i] = a
   end do
end if
sync all
temp = temp + this_image()
```

Synchronization

- **We have to be careful with one-sided updates**
  - If we get remote data was it valid?
  - Could another process send us data and overwrite something we have not yet used?
  - How do we know when remote data has arrived?
- **The standard introduces execution segments to deal with this, segments are bounded by image control**
- **If a non-atomic variable is defined in a segment, it must not be referenced, defined, or become undefined in a another segment unless the segments are ordered**

Execution segments



Recap of coarray basics

- Multiple images execute asynchronously
- We can declare a coarray which is accessible from remote images
- Indexing with [ ] is used to access remote data
- We can find out which image we are
  - num_images()
  - this_image()
- We can synchronize to make sure variables are up to date
  - sync all
- Now consider a program example…

Example: Calculate density of primes

- Use function num_primes on each image

```
program pdensity
 implicit none
 integer, parameter :: n=10000000
 integer start,end,i
 integer, dimension(:)[:], allocatable :: nprimes
 real density

 allocate( nprimes(num_images())[*] )

 start = (this_image()-1) * n/num_images() + 1
 end = start + n/num_images() - 1

 nprimes(this_image())[1] = num_primes(start,end)

 sync all
```

Example: Calculate density of primes, cont.

- Image #1 gets values from other images, then computes and prints prime density

```
if (this_image()==1) then

  nprimes(1)=sum(nprimes)

  density=real(nprimes(1))/n

  print *,"Calculating prime density on", &
&         num_images(),"images"
  print *,nprimes(1),'primes in',n,'numbers'
  write(*,'(" density is ",2Pf0.2,"%")')density
  write(*,'(" asymptotic theory gives ", &
&           2Pf0.2,"%")')1.0/(log(real(n))-1.0)

end if
```

Example: Calculate density of primes, cont.

```
Calculated prime density on 16 images
664580 primes in 10000000 numbers
density is 6.65%
asymptotic theory gives 6.61%
Done in 2.86 seconds
```

- Did anyone spot an error in the first program slide ?
- Try it in the lab session

Multi-codimensional arrays

- A coarray can have multiple codimensions

```
complex :: b[0:*]
complex :: p(32,32)[2,3,*]
```

- Cosubscripts are mapped to images according to Fortran array-element order

image	b(:)[i]	p(:)[i,j,k]
1	b(:)[0]	p(:)[1,1,1]
2	b(:)[1]	p(:)[2,1,1]
3	b(:)[2]	p(:)[1,2,1]
4	b(:)[3]	p(:)[2,2,1]
5	b(:)[4]	p(:)[1,3,1]
6	b(:)[5]	p(:)[2,3,1]
7	b(:)[6]	p(:)[1,1,2]

Multi-codimensional arrays, cont.

- There is a way to find out which part of the coarray is mapped to an image
  - `this_image(coarray)` yields codimensions
  - `this_image(coarray,dim)` yields one codimension
- So for the previous example, on image 2
  - `this_image(p)` is [2, 1, 1]
- Can get image index from coarray:
  - `image_index(p,[2,1,1])` is 2
  - `image_index(p,[3,4,2])` is 0 since [3,4,2] is not a valid set of cosubscripts

Allocatable coarrays

```
integer n,ni
real, allocatable :: pmax(:)[:]
real, allocatable :: p(:,:)[:,:]

ni = num_images()
allocate( pmax(ni)[*], p(n,n)[4,*] )
```

- Require same shape and coshape on every image
  - Last codimension must always be unspecified
- Allocate and deallocate with coarray arguments cause a synchronization

Adding coarray with minimal code changes

- Coarray structures can be inserted into an existing MPI code to simplify and improve communication without affecting the general code structure
- In some cases the usage of FORTRAN pointers can simplify the introduction of coarray variables in a routine without modifying the calling tree
- Code modifications are mostly limited to the routine where coarrays must be introduced
- A coarray is not permitted to be a pointer: however, a coarray may be of a derived type with pointer or allocatable components

Pointer Coarray Structure Components

- **We are allowed to have a coarray that contains components that are pointers**
- **Note that the pointers have to point to local data**
- **We can then access one of the pointers on a remote image to get at the data it points to**
- **This technique is useful when adding coarrays into an existing MPI code**
  - We can insert coarray code deep in call tree without changing many subroutine argument lists
  - We don’t need new coarray declarations

Example: adding coarrays to existing code

- Existing non-coarray arrays u,v,w
- Create a type (coords) to hold pointers (x,y,z) that we use to point to x,y,z. We can use the vects coarray to access u, v, w.

```
subroutine calc(u,v,w)
real, intent(in), target, dimension(100) :: u,v,w
type coords
      real, pointer, dimension(:) :: x,y,z
end type coords
type(coords), save :: vects[*]
! …
vects%x => u ; vects%y => v ; vects%z => w
sync all
firstx = vects[1]%x(1)
```

Features we won't cover

- **Memory synchronization (sync memory)**
  - completion of remote operations but not segment ordering
- **critical section (critical , … , end critical)**
  - only one image executes the section at a time
- **locks**
  - control access to data held by one image
- **status and error conditions for image control**
- **atomic subroutines (useful for flag variables)**
- **I/O**

Example: distributed remote gather

- The problem is how to implement the following gather loop on a distributed memory system

```
REAL     :: table(n), buffer(nelts)
INTEGER :: index(nelts)    ! nelts << n
...
DO i = 1, nelts
  buffer(i) = table(index(i))
ENDDO
```

- The array table is distributed across the processors, while index and buffer are replicated
- Synthetic code, but simulates “irregular” communication access patterns

Remote gather: MPI implementation

- MPI rank 0 controls the index and receives the values from the other ranks

```
IF (mype.eq.0)THEN

   isum=0
!  PE0 gathers indices to send out to individual PEs
   DO i=1,nelts
      pe =(index(i)-1)/nloc
      isum(pe)=isum(pe)+1
      who(isum(pe),pe) = index(i)
   ENDDO
!  send out count and indices to PEs
   DO i = 1, npes-1
      CALL MPI_SEND(isum(i),1,MPI_INTEGER,i,10.
      IF(isum(i).gt.0)THEN
         CALL MPI_SEND(who(1,i),isum(i),...
      ENDIF
   ENDDO
!  now wait to receive values and scatter them.
   DO i = 1,isum(0)
      offset = mod(who(i,0)-1,nloc)+1
      buff(i,0) = mpi_table(offset)
   ENDDO
   DO i = 1,npes-1
      IF(isum(i).gt.0)THEN
         CALL MPI_RECV(buff(1,i),isum(i),...
      ENDIF
   ENDDO
   DO i=nelts,1,-1
      pe =(index(i)-1)/nloc
      offset = isum(pe)
      mpi_buffer(i) = buff(offset,pe)
      isum(pe) = isum(pe) - 1
   ENDDO

ELSE  !IF my_rank.ne.0

!  Each PE gets the list and sends the values to PE0

   CALL MPI_RECV(my_sum,1,MPI_INTEGER,...
   IF(my_sum.gt.0)THEN
      CALL MPI_RECV(index,my_sum,MPI_INTEGER,...
      DO i = 1, my_sum
         offset = mod(index(i)-1,nloc)+1
         mpi_buffer(i) = mpi_table(offset)
      ENDDO
      CALL MPI_SEND(mpi_buffer,my_sum,...
   ENDIF

ENDIF
```

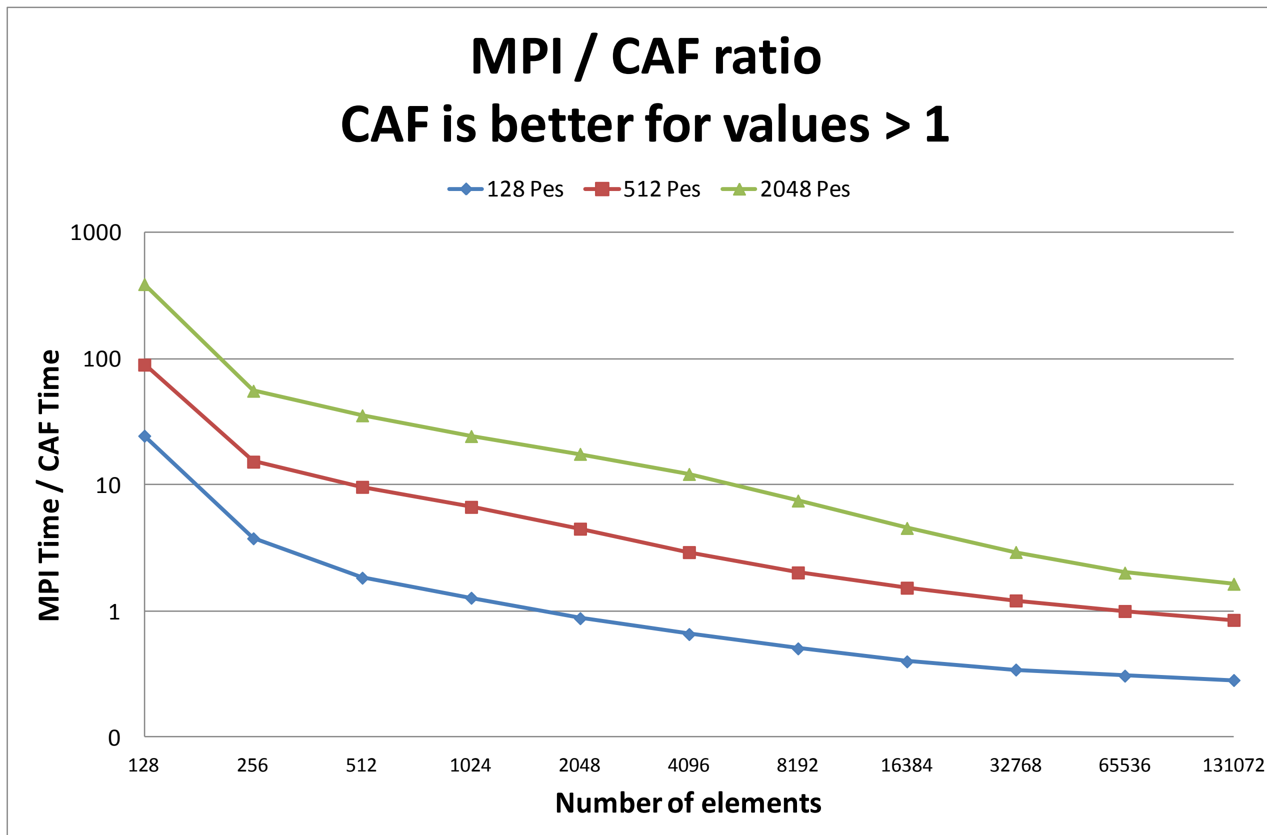
Remote gather: coarray implementation (get)

- Image 1 gets the values from the other images

```
IF (myimg.eq.1) THEN
    DO i=1,nelts
       pe =(index(i)-1)/nloc+1
       offset = MOD(index(i)-1,nloc)+1
       caf_buffer(i) = caf_table(offset)[pe]
    ENDDO
 ENDIF
```

Remote gather results

- Coarray implementations are much simpler
- Coarray syntax allows the expression of remote data in a natural way – no need of complex protocols



Future for coarrays in Fortran

- Additional coarray features may be described in a Technical Specification (TS)

- Developed as part of the official ISO standards process

- Work in progress and the areas of discussion are:
  - image teams
  - collective intrinsics for coarrays
    - `CO_BCAST, CO_SUM, CO_MAX, CO_REDUCE`
  - atomics

References

- ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
  Coarrays in the next Fortran Standard
  John Reid, April 2010

- http://lacsi.rice.edu/software/caf/downloads/documentation/nrRAL98060.pdf
  Co-array Fortran for parallel programming
  Numrich and Reid, 1998

- Ashby, J.V. and Reid, J.K (2008). Migrating a scientific application from MPI to coarrays.
  CUG 2008 Proceedings. RAL-TR-2008-015
  See http://www.numerical.rl.ac.uk/reports/reports.shtml

UPC

UPC Overview and Design Philosophy

- **Unified Parallel C (UPC) is:**
  - An explicit parallel extension of ANSI C
  - A partitioned global address space language
  - Sometimes called a GAS language
- **Similar to the C language philosophy**
  - Programmers are clever and careful, and may need to get close to hardware
    - to get performance, but
    - can get in trouble
  - Concise and efficient syntax
- **Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C**
- **Based on ideas in Split-C, AC, and PCP**

UPC Execution Model

- **A number of threads working independently in a SPMD fashion**
  - Number of threads specified at compile-time or run-time; available as program variable `THREADS`
  - `MYTHREAD` specifies thread index (`0..THREADS-1`)
  - `upc_barrier` is a global synchronization: all wait
  - There is a form of parallel loop, `upc_forall`
- **There are two compilation modes**
  - Static Threads mode:
    - `THREADS` is specified at compile time by the user
    - The program may use `THREADS` as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

Hello World in UPC

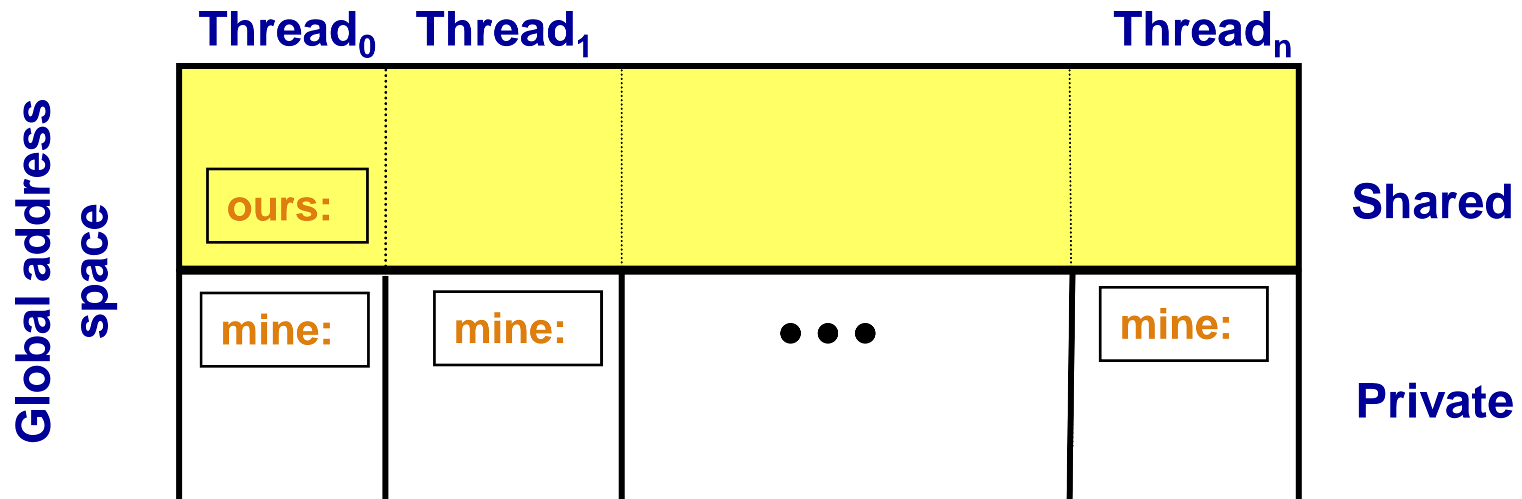
- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the identifiers from the previous slides, we can parallel hello world:

```
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

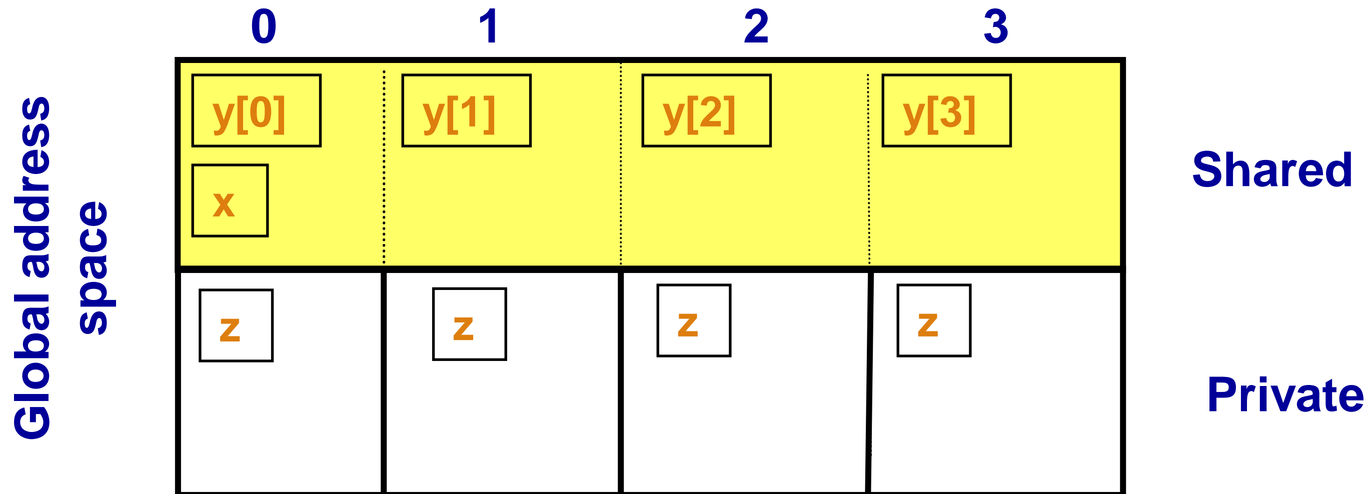
```
shared int ours;  // use sparingly: performance
int mine;
```



Shared and Private Data

- **Examples of Shared and Private Data Layout:**
  - Assume THREADS = 4

```
shared int x;  /* x will have affinity to thread 0 */
shared int y[THREADS];
int z;
```



Shared and Private Data

- Shared Data Layout

```
shared int A[4][THREADS];
```

Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

Blocking of shared data

- Default block size is 1
- Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.
- A block size is specified in the declaration as follows:

```
shared [block-size] type array[N];
shared [4] int a[16];
```

- Block size and THREADS determine affinity
- The term affinity means in which thread's local shared-memory space, a shared data item will reside
- Element i of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \bmod THREADS$$

Shared and Private Data

- **Can become dangerous**
  - Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

upc_forall

- A vector addition can be written as follows…

```
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];

void main() {
  int i;
  upc_forall(i=0; i<N; i++; i)
    sum[i]=v1[i]+v2[i];
}
```

- upc_forall adds an extra argument for affinity control
- The code would be correct but slow if the affinity expression were i+1 rather than i.

UPC pointers

```
int *p1;                /* private pointer
                           to local memory */
shared int *p2;         /* private pointer
                           to shared space */
int *shared p3;         /* shared pointer
                           to local memory */
shared int *shared p4; /* shared pointer
                           to shared space */
```

Where does the pointer point?

Where does the pointer reside?

	Local	Shared
Private	PP (p1)	PS (p2)
Shared	~~SP (p3)~~	SS (p4)

Shared to private is not recommended

Common Uses for UPC Pointer Types

- **int *p1;**
  - These pointers are fast (just like C pointers)
  - Use to access local data in part of code performing local work
  - Often cast a pointer-to-shared to one of these to get faster access to shared data that is local
- **shared int *p2;**
  - Use to refer to remote data
  - Larger and slower due to test-for-local + possible communication
- **int *shared p3;**
  - Not recommended
- **shared int *shared p4;**
  - Use to build shared linked structures, e.g., a linked list

Dynamic Memory Allocation in UPC

- Dynamic memory allocation of shared memory is available in UPC
- Functions can be collective or not
- A collective function has to be called by every thread and will return the same value to all of them
- As a convention, the name of a collective function typically includes “all”

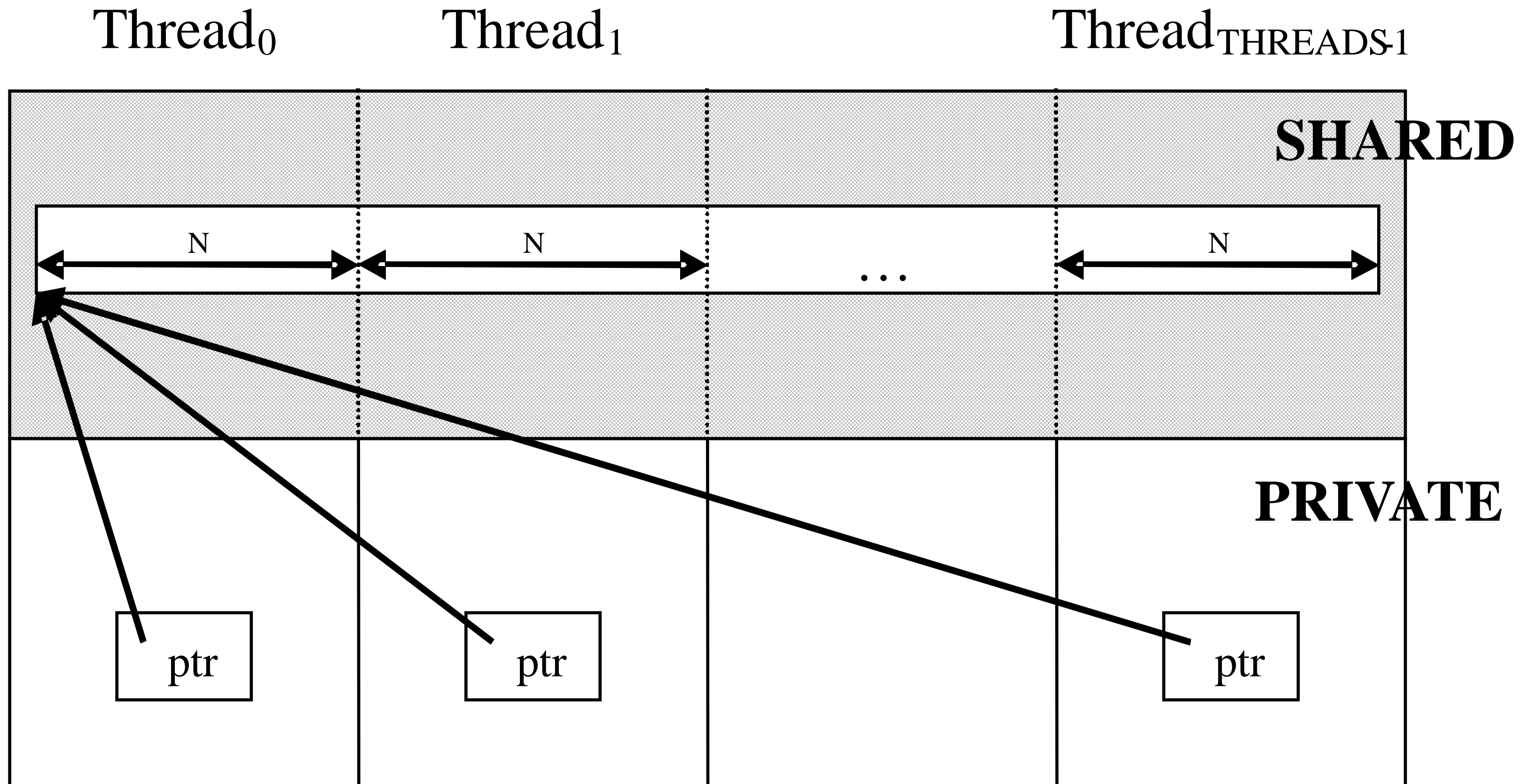
Collective Global Memory Allocation

- upc_all_alloc

```
shared void *upc_all_alloc
                (size_t nblocks, size_t nbytes);
nblocks: number of blocks
nbytes:    block size
```

- This function has the same result as upc_global_alloc. But this is a collective function, which is expected to be called by all threads
- All the threads will get the same pointer
- Equivalent to :
  `shared [nbytes] char[nblocks * nbytes]`

Collective Global Memory Allocation




```
shared [N] int *ptr;
ptr = (shared [N] int *)
        upc_all_alloc( THREADS, N*sizeof( int ) );
```

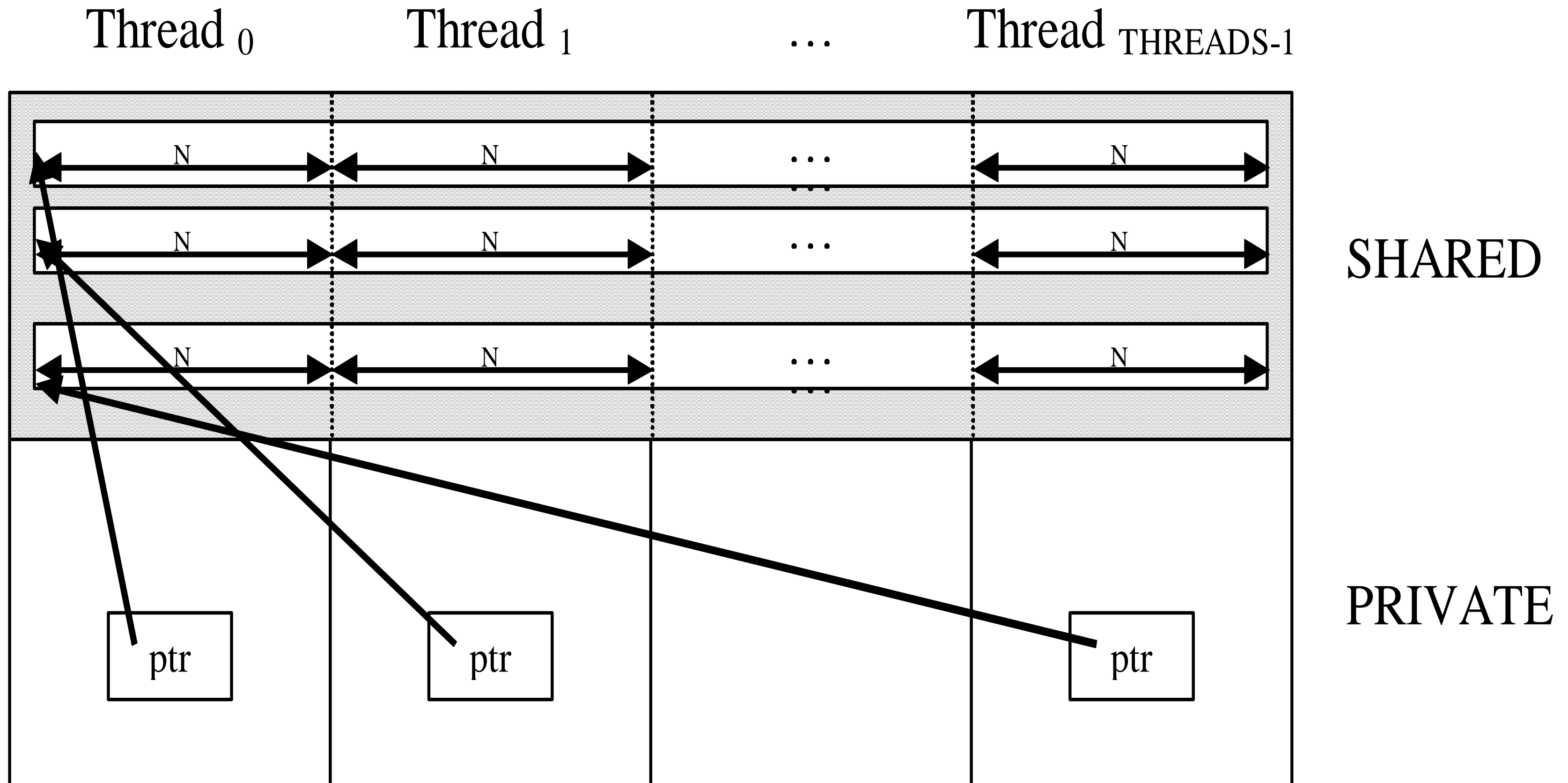
Global Memory Allocation

- upc_global_alloc

```
shared void *upc_global_alloc
    (size_t nblocks, size_t nbytes);
nblocks : number of blocks
nbytes : block size
```

- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory region in the shared space
- Space allocated per calling thread is equivalent to :
  `shared [nbytes] char[nblocks * nbytes]`
- If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer

upc_global_alloc



Local-Shared Memory Allocation

CRAY

- upc_alloc

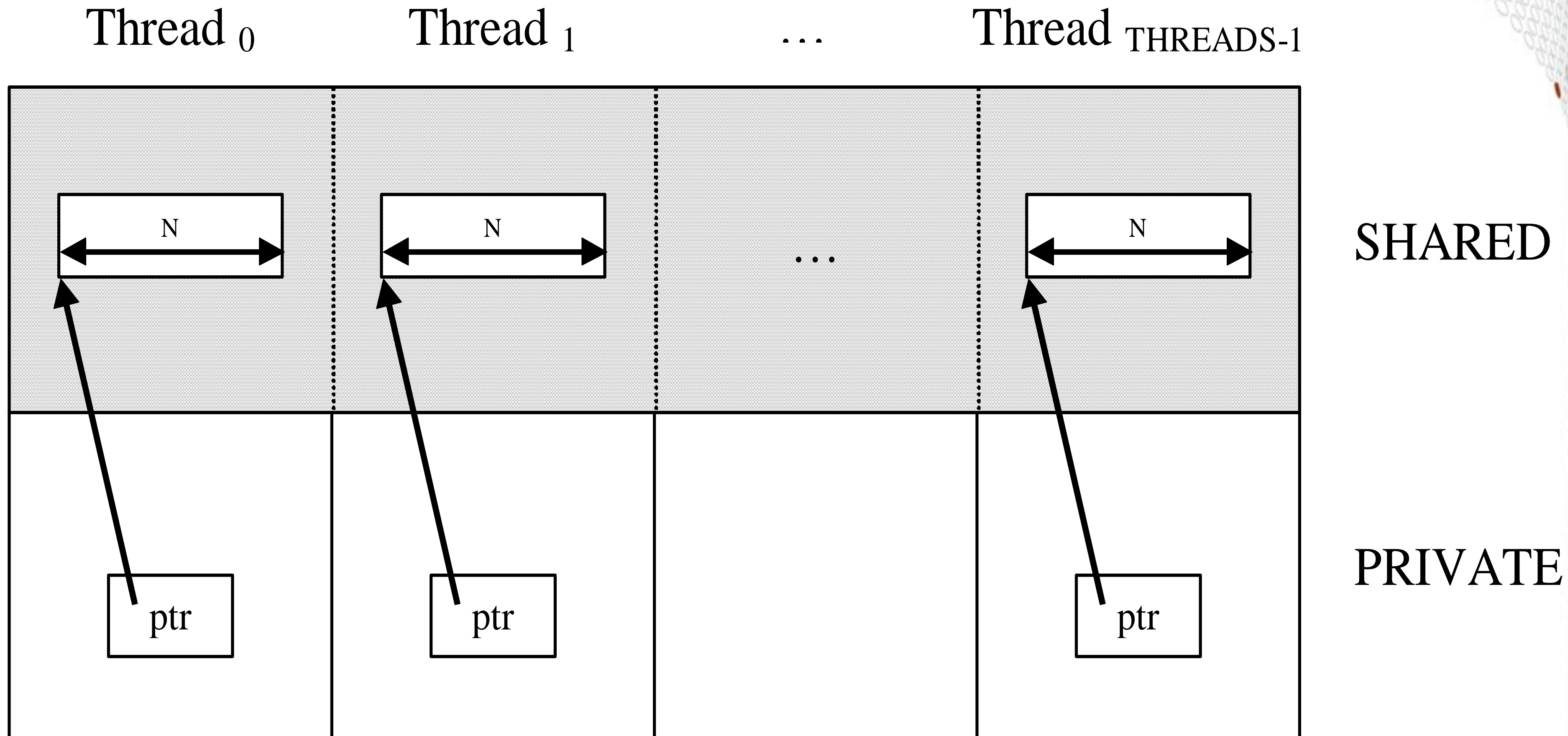
```
shared void *upc_alloc (size_t nbytes);
```

```
nbytes: block size
```

- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory region in the local-shared space of the calling thread
- Space allocated per calling thread is equivalent to :
  `shared [] char[nbytes]`

- If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer

Local-Shared Memory Allocation




```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof( int ));
```

Memory Space Clean-up

- upc_free

```
void upc_free(shared void *ptr);
```

- The upc_free function frees the dynamically allocated shared memory pointed to by ptr

- upc_free is not collective

Features we won't cover

- **Synchronization – no implicit synchronization among the threads – it's up to you!**
  - Barriers (Blocking)
  - Split-Phase Barriers (Non-blocking)
  - Locks – collective and global
- **String functions in UPC**
  - UPC equivalents of memcpy, memset
- **Special functions**
  - Shared pointer information (phase, block size, thread number)
  - Shared object information (size, block size, element size)
- **UPC collectives**
- **UPC-IO**

References

- http://upc.gwu.edu/
  Unified Parallel C at George Washington University
- http://upc.lbl.gov/
  Berkeley Unified Parallel C Project
- http://docs.cray.com/
  Cray C and C++ Reference Manual

Cray programming environment and PGAS compilers

Roberto Ansaloni

Compiling, Linking and Launching a CAF code

- Load Cray programming environment (if not default)

  `module load PrgEnv-cray`

- The caf compiler option to enable recognition of Coarray syntax: `-h caf`
  - Will be default in next CCE release

- Compile and run

```
ftn -hcaf hello.c -o hello
aprun -n <npes> ./hello
```

Compiling, Linking and Launching a UPC code

- Load Cray programming environment (if not default)
- module load PrgEnv-cray

- The upc compiler option:

```
-h upc        ( enable recognition of UPC syntax)
-X <npes>     ( optional to statically set THREADS constant)
```

- Compile and run

```
cc -hupc hello.c -o hello
aprun -n <npes> ./hello
```

- If –X npes is used at compile time, you must specify the same number of threads in the aprun command.

Symmetric Heap

- By default, each PE reserves 64 MB of symmetric heap space. To increase or decrease this amount, set the `XT_SYMMETRIC_HEAP_SIZE` environment variable (Use suffixes K, M, and G)

```
export XT_SYMMETRIC_HEAP_SIZE=512M
```

Symmetric Heap and Huge pages on Cray XE6

- The symmetric heap is mapped onto hugepages by DMAPP. It is advisable to also map the static data and/or private heap onto huge pages.
- If huge pages are not used, remotely mapped memory is limited to 2GB per node
- Several sizes available
  - 128K, 512K, 2M, 8M, 16M, 64M

```
module load craype-hugepages2M
```

- More info: `man intro_hugepages`

pgas defer_sync directive

- The compiler synchronizes the references in a statement as late as possible without violating program semantics.
- The purpose of the defer_sync directive is to synchronize the references even later, beyond where the compiler can determine it is safe.
- The programmer is responsible for inserting the proper synchronization at the right time
- This can be used to overlap remote memory references to other operations

```
#pragma pgas defer_sync
!DIR$ PGAS DEFER_SYNC
```

Useful man pages

- **man intro_pgas**
  - General info about Cray CCE PGAS support
- **man defer_sync**
  - Some more info and an example
- **man directives**
  - Introduction to Cray C/C++ compiler #pragmas and Cray Fortran Compiler directives
  - Different types: General, Vectorization, Scalar, Inlining, PGAS

Asynchronous MPI communication

- **Effective MPI communication/computation overlap requires Progress Engine Support**
  - This is automatically implemented in MPT 5.4.0 by helper threads that progress the MPI state engine while application is computing
- **Only effective if used with core specialization to reserve a core/node for the helper threads**
  - It is likely to have spare CPU cores on the node in a GPU code
- **Must set the following variables to enable it**

```
export MPICH_NEMESIS_ASYNC_PROGRESS=1
export MPICH_MAX_THREAD_SAFETY=multiple
```

- **Limit the OpenMP parallelism**

```
export OMP_NUM_THREADS=14
```

- **Run the application with core specialization**

```
aprun -n XX -d 14 -r 2 ./a.out
```

Node ordering

- The set of nodes assigned to an application is chosen by "the system"
  - Cray ALPS, batch system

- The only thing a user can do is modify the mapping between MPI ranks and the given nodes

- The way to do this is by using MPICH rank reordering

Set the preferred node order into file `MPICH_RANK_ORDER`

```
export MPICH_RANK_REORDER_METHOD=3
```

- More info: mpi man page
  - look for MPICH_RANK_REORDER_METHOD

grid_order

- The grid_order tool can help to properly set `MPICH_RANK_ORDER` in case of specific topologies
  - Successfully used with CP2K
- The tool is provided by Cray perftools: either load the module or set PATH to use it

```
module load perftools
export PATH=$PATH:opt/cray/perftools/default/bin
```

- Examples

```
grid_order -R -c 1,1 -g m,m -H >MPICH_RANK_ORDER
grid_order -R -c ppn,nth -g npx,npy -H >MPICH_RANK_ORDER
```

- For more info:

```
grid_order -h
```

Cray performance tools for MPI and PGAS code development and tuning

Roberto Ansaloni

Cray perftools design goals

- **Assist the user with application performance analysis and optimization**
  - Help user identify important and meaningful information from potentially massive data sets
  - Help user identify problem areas instead of just reporting data
  - Bring optimization knowledge to a wider set of users
- **Focus on ease of use and intuitive user interfaces**
  - Automatic program instrumentation
  - Automatic analysis
- **Target scalability issues in all areas of tool development**
  - Data management
  - Storage, movement, presentation

Cray perftools features

- **Provide a complete solution from instrumentation to measurement to analysis to visualization of data**
- **Performance measurement and analysis on large systems**
  - Automatic Profiling Analysis
  - Load Imbalance
  - HW counter derived metrics
  - Predefined trace groups provide performance statistics for libraries called by program (blas, lapack, pgas runtime, netcdf, hdf5, etc.)
  - Observations of inefficient performance
  - Data collection and presentation filtering
  - Data correlates to user source (line number info, etc.)
  - Support MPI, SHMEM, OpenMP, UPC, CAF
  - Access to network counters
  - Minimal program perturbation

The Cray Performance Analysis Framework

CRAY

- **Supports traditional post-mortem performance analysis**
  - Automatic identification of performance problems
  - Indication of causes of problems
  - Suggestions of modifications for performance improvement
- **pat_build**
  - provides automatic instrumentation
- **CrayPat run-time library**
  - collects measurements (transparent to the user)
- **pat_report**
  - performs analysis and generates text reports
- **pat_help**
  - online help utility
- **Cray Apprentice^2**
  - graphical visualization tool

Collecting Performance Data

- **Sampling**
  - External agent (asynchronous)
  - Timer interrupt
  - Hardware counters overflow
- **Tracing**
  - Internal agent (synchronous)
  - Code instrumentation
  - Automatic or manual instrumentation
- **While event tracing provides most useful information, it can be very heavy if the application runs on a large number of cores for a long period of time**
- **Sampling can be useful as a starting point, to provide a first overview of the work distribution**

Application instrumentation with pat_build

- `pat_build` is a stand-alone utility that automatically instruments the application for performance collection
  ```
  module load perftools
  ```
- Requires no source code or makefile modification
  - Automatic instrumentation at group (function) level
  - Groups: mpi, io, heap, math SW, …
- Performs link-time instrumentation
  - **Requires object files**
  - Instruments optimized code
  - Generates stand-alone instrumented program
  - Preserves original binary

Automatic Profiling Analysis (APA)

- Provides simple procedure to instrument and collect performance data for novice users
- Identifies top time consuming routines
- Automatically creates instrumentation template customized to application for future in-depth measurement and analysis

```
pat_build –Oapa a.out
```

Program Instrumentation

- **Large programs: 2-step approach**
  - Scaling issues more dominant
  - Use automatic profiling analysis to quickly identify top time consuming routines
  - Use loop statistics to quickly identify top time consuming loops
  - Run tracing experiments on a selected number of routines

```
$ pat_build –Oapa a.out
$ aprun a.out+pat
$ pat_build –T <sub1,sub2...> -g mpi a.out
$ aprun a.out+pat
```

- **Small (test) or short running programs**
  - Scaling issues not significant
  - Can skip first sampling experiment and directly generate profile

```
$ pat_build –u –g mpi a.out
$ aprun a.out+pat
```

Steps to Collecting Performance Data (1/3) Instrument the code and run 1st sampling test

- **Access performance tools software**

```
$ module load perftools
```

- **Build application keeping .o files (CCE: -h keepfiles)**

```
$ make clean ; make
```

- **Instrument application for automatic profiling analysis**
  - You should get an instrumented program a.out+pat

```
$ pat_build –O apa a.out
```

- **Run application to get top time consuming routines**
  - You should get a performance file (“<sdatafile>.xf”) or multiple files in a directory (<sdatadir>)

```
$ aprun … a.out+pat
```

Steps to Collecting Performance Data (2/3
Generate 1st report and APA file

- Generate report and .apa instrumentation file
  - You should get an APA file .apa

```
$ pat_report -o <samprpt> [<sdatafile>.xf | <sdatadir>]
```

- Inspect `.apa` file and sampling report `<samprpt>`
- Verify if additional instrumentation is needed

APA File Example

```
#  You can edit this file, if desired, and use it
#   to reinstrument the program for  tracing like this:
#
#           pat_build -O standard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-
5.0.0.2-
Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=none.
14999.xf.xf.apa
#
#  These suggested trace options are based on data from:
#
#
/home/users/malice/pat/Runs/Runs.seal.pat5001.2009Apr04/./pat.quad/homme/stan
dard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2-
Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=none.
14999.xf.xf.cdb
# ----------------------------------------------------------------------

#       HWPC group to collect by default.

  -Drtenv=PAT_RT_HWPC=1  # Summary with TLB metrics.

# ----------------------------------------------------------------------

#       Libraries to trace.

  -g mpi

# ----------------------------------------------------------------------

#       User-defined functions to trace, sorted by % of samples.

#       The way these functions are filtered can be controlled with
#       pat_report options (values used for this file are shown):
#
#       -s apa_max_count=200    No more than 200 functions are listed.
#       -s apa_min_size=800     Commented out if text size < 800 bytes.
#       -s apa_min_pct=1        Commented out if it had < 1% of samples.
#       -s apa_max_cum_pct=90   Commented out after cumulative 90%.

#       Local functions are listed for completeness, but cannot be traced.

  -w  # Enable tracing of user-defined functions.
      # Note: -u should NOT be specified as an additional option.
```

```
# 31.29%  38517 bytes
    -T prim_advance_mod_preq_advance_exp_

# 15.07%  14158 bytes
    -T prim_si_mod_prim_diffusion_

#  9.76%  5474 bytes
    -T derivative_mod_gradient_str_nonstag_

...

#  2.95%  3067 bytes
    -T forcing_mod_apply_forcing_

#  2.93%  118585 bytes
    -T column_model_mod_applycolumnmodel_

#  Functions below this point account for less than 10% of samples.

#  0.66%  4575 bytes
#      -T bndry_mod_bndry_exchangev_thsave_time_

#  0.10%  46797 bytes
#      -T baroclinic_inst_mod_binst_init_state_

#  0.04%  62214 bytes
#      -T prim_state_mod_prim_printstate_

...
#  0.00%  118 bytes
#      -T time_mod_timelevel_update_

# ----------------------------------------------------------------------

 -o preqx.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x+apa
# New instrumented program.

 /.AUTO/cray/css.pe_tools/malice/craypat/build/pat/2009Apr03/2.1.56HD/amd64/homme/pgi/pat-
5.0.0.2/homme/2005Dec08/build.Linux/preqx.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x  #
Original program.
```

Steps to Collecting Performance Data (3/3) Generating profile from APA

- Instrument application for further analysis (a.out+apa)

```
$ pat_build -O <apafile>.apa
```

- Run application

```
$ aprun … a.out+apa
```

- Generate text report and visualization file (.ap2)

```
$ pat_report -o <tracrpt> [<datafile>.xf | <datadir>]
```

- View report in text and/or with Cray Apprentice2

```
$ app2 <datafile>.ap2
```

Files Generated and the Naming Convention

File Suffix	Description
a.out+pat	Program instrumented for data collection
a.out…s.xf	Raw data for sampling experiment, available after application execution
a.out…t.xf	Raw data for trace (summarized or full) experiment, available after application execution
a.out…st.ap2	Processed data, generated by pat_report, contains application symbol information
a.out…s.apa	Automatic profiling pnalysis template, generated by pat_report (based on pat_build –O apa experiment)
a.out+apa	Program instrumented using .apa file
MPICH_RANK_ORDER.Custom	Rank reorder file generated by pat_report from automatic grid detection an reorder suggestions

Some further details

- **The application must run on Lustre (/scratch/…) to provide the profile data**
  - Can be customized with `PAT_RT_EXPFILE_MAX`
- **It is useful to save the .ap2 file**
  - The ".ap2" file is a self contained compressed performance file
  - Normally it is about 5 times smaller than the ".xf" file
  - Contains the information needed from the application binary
  - Can be reused, even if the application binary is no longer available or if it was rebuilt
  - It is the only input format accepted by Cray Apprentice^2
- **PAT_RT_XXX environment variables**
  - Control perftools runtime
  - See intro_craypat man page
  - Enable collection of HW counters

pat_report

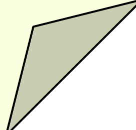
- **Performs data conversion**
  - Combines information from binary with raw performance data
- **Performs analysis on data**
- **Generates text report of performance results**
- **Formats data for input into Cray Apprentice^2**

Job Execution Information

```
CrayPat/X:  Version 5.2.3.8078 Revision 8078 (xf 8063)  08/25/11 …

Number of PEs (MPI ranks):   16

Numbers of PEs per Node:     16

Numbers of Threads per PE:    1

Number of Cores per Socket:  12

Execution start time:  Thu Aug 25 14:16:51 2011

System type and speed:  x86_64 2000 MHz

Current path to data file:
  /lus/scratch/heidi/ted_swim/mpi-openmp/run/swim+pat+27472-34t.ap2

Notes for table 1:
…
```

Sampling Output (Table 1)

```
Notes for table 1:

...

Table 1:  Profile by Function

 Samp %  | Samp  |  Imb.  |   Imb.   |Group
         |       |  Samp  | Samp %   | Function
         |       |        |          |  PE='HIDE'

 100.0%  |  775  |    --  |     --   |Total
|------------------------------------------
|  94.2% |  730  |    --  |     --   |USER
||-----------------------------------------
||  43.4% |  336  |  8.75  |   2.6%  |mlwxyz_
||  16.1% |  125  |  6.28  |   4.9%  |half_
||   8.0% |   62  |  6.25  |   9.5%  |full_
||   6.8% |   53  |  1.88  |   3.5%  |artv_
||   4.9% |   38  |  1.34  |   3.6%  |bnd_
||   3.6% |   28  |  2.00  |   6.9%  |currenf_
||   2.2% |   17  |  1.50  |   8.6%  |bndsf_
||   1.7% |   13  |  1.97  |  13.5%  |model_
||   1.4% |   11  |  1.53  |  12.2%  |cfl_
||   1.3% |   10  |  0.75  |   7.0%  |currenh_
||   1.0% |    8  |  5.28  |  41.9%  |bndbo_
||   1.0% |    8  |  8.28  |  53.4%  |bndto_
||=========================================
|   5.4% |   42  |    --  |     --   |MPI
||-----------------------------------------
||   1.9% |   15  |  4.62  |  23.9%  |mpi_sendrecv_
||   1.8% |   14  | 16.53  |  55.0%  |mpi_bcast_
||   1.7% |   13  |  5.66  |  30.7%  |mpi_barrier_
|==========================================
```



pat_report: Flat Profile

```
Table 1:  Profile by Function Group and Function

 Time % |        Time |Imb. Time |   Imb.  | Calls |Group
        |             |          | Time %  |       | Function
        |             |          |         |       |  PE='HIDE'

 100.0% | 104.593634 |       -- |      -- | 22649 |Total
|------------------------------------------------------------
|  71.0% |  74.230520 |       -- |      -- | 10473 |MPI
||-----------------------------------------------------------
||  69.7% |  72.905208 | 0.508369 |   0.7% |   125 |mpi_allreduce_
||   1.0% |   1.050931 | 0.030042 |   2.8% |    94 |mpi_alltoall_
||===========================================================
|  25.3% |  26.514029 |       -- |      -- |    73 |USER
||-----------------------------------------------------------
||  16.7% |  17.461110 | 0.329532 |   1.9% |    23 |selfgravity_
||   7.7% |   8.078474 | 0.114913 |   1.4% |    48 |ffte4_
||===========================================================
|   2.5% |   2.659429 |       -- |      -- |   435 |MPI_SYNC
||-----------------------------------------------------------
||   2.1% |   2.207467 | 0.768347 |  26.2% |   172 |mpi_barrier_(sync)
||===========================================================
|   1.1% |   1.188998 |       -- |      -- | 11608 |HEAP
||-----------------------------------------------------------
||   1.1% |   1.166707 | 0.142473 |  11.1% |  5235 |free
|============================================================
```

pat_report: Message Stats by Caller

```
Table 4:  MPI Message Stats by Caller

    MPI Msg |MPI Msg |  MsgSz |  4KB<= |Function
      Bytes |  Count |   <16B |  MsgSz | Caller
            |        |  Count |  <64KB |  PE[mmm]
            |        |        |  Count |

 15138076.0 | 4099.4 |  411.6 | 3687.8 |Total
|-------------------------------------------------
| 15138028.0 | 4093.4 |  405.6 | 3687.8 |MPI_ISEND
||-----------------------------------------------
||  8080500.0 | 2062.5 |   93.8 | 1968.8 |calc2_
3|            |        |        |        | MAIN_
||||---------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  8208000.0 | 2000.0 |     -- | 2000.0 |pe.9
4|||  6160000.0 | 2000.0 |  500.0 | 1500.0 |pe.15
||||=============================================
||  6285250.0 | 1656.2 |  125.0 | 1531.2 |calc1_
3|            |        |        |        | MAIN_
||||---------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.3
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.5
||||=============================================
.  .  .
```

pat_report: MPI Message Stats by Caller

```
Table 4:  MPI Message Stats by Caller

    MPI Msg |MPI Msg |  MsgSz |  4KB<= |Function
      Bytes |  Count |   <16B | MsgSz | Caller
            |        |  Count | <64KB |  PE[mmm]
            |        |        | Count |

 15138076.0 | 4099.4 |  411.6 | 3687.8 |Total
|-------------------------------------------------
| 15138028.0 | 4093.4 |  405.6 | 3687.8 |MPI_ISEND
||------------------------------------------------
||  8080500.0 | 2062.5 |   93.8 | 1968.8 |calc2_
3|            |        |        |        | MAIN_
||||----------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  8208000.0 | 2000.0 |     -- | 2000.0 |pe.9
4|||  6160000.0 | 2000.0 |  500.0 | 1500.0 |pe.15
||||==============================================
||  6285250.0 | 1656.2 |  125.0 | 1531.2 |calc1_
3|            |        |        |        | MAIN_
||||----------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.3
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.5
||||==============================================
.  .  .
```

Automatic Communication Grid Detection

- Analyze runtime performance data to identify grids in a program to maximize on-node communication
- Determine whether or not a custom MPI rank order will produce a significant performance benefit
- Grid detection is helpful for programs with significant point-to-point communication
- Tools produce a custom rank order if it's beneficial based on grid size, grid order and cost metric
- Available if MPI functions traced (-g mpi)
- Describe how to re-run with custom rank order

MPI grid detection report

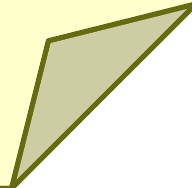
```
MPI Grid Detection:  There appears to be point-to-point MPI
     communication in a 22 X 18 grid pattern. The 48.6% of the total
     execution time spent in MPI functions might be reduced with a rank
     order that maximizes communication between ranks on the same node.
     The effect of several rank orders is estimated below.

     A file named MPICH_RANK_ORDER.Custom was generated along with this
     report and contains the Custom rank order from the following table.
     This file also contains usage instructions and a table of
     alternative rank orders.

          Rank     On-Node      On-Node   MPICH_RANK_REORDER_METHOD
         Order    Bytes/PE    Bytes/PE%
                               of Total
                               Bytes/PE

         Custom   7.80e+06       78.37%   3
            SMP   5.59e+06       56.21%   1
           Fold   2.59e+05        2.60%   2
     RoundRobin   0.00e+00        0.00%   0
```

PGAS Support

- **Profiles of a PGAS program can be created to show:**
  - Top time consuming functions/line numbers in the code
  - Load imbalance information
  - Performance statistics attributed to user source by default
  - Can expose statistics by library as well
  - To see underlying operations, such as wait time on barriers
- **Data collection is based on methods used for MPI library**
  - PGAS data is collected by default when using Automatic Profiling Analysis (pat_build –O apa)
  - Predefined wrappers for runtime libraries (caf, upc, pgas) enable attribution of samples or time to user source
- **UPC and SHMEM heap tracking available**
  - -g heap will track shared heap in addition to local heap

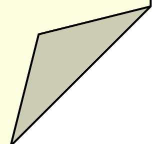
PGAS Report Showing Library Functions

```
Table 1:  Profile by Function and Callers, with Line Numbers
 Samp % | Samp |Group
        |      | Function
        |      |  Caller
        |      |   PE='HIDE'
 100.0% |   47 |Total
|--------------------------
|  93.6% |   44 |ETC
||-------------------------
||  85.1% |   40 |upc_memput
3|        |      | all2all:mpp_bench.c:line.298
4|        |      |  do_all2all:mpp_bench.c:line.348
5|        |      |   main:test_all2all.c:line.70
||   4.3% |    2 |bzero
3|        |      | (N/A):(N/A):line.0
||   2.1% |    1 |upc_all_alloc
3|        |      | mpp_alloc:mpp_bench.c:line.143
4|        |      |  main:test_all2all.c:line.25
||   2.1% |    1 |upc_all_reduceUL
3|        |      | mpp_accum_long:mpp_bench.c:line.185
4|        |      |  do_cksum:mpp_bench.c:line.317
5|        |      |   do_all2all:mpp_bench.c:line.341
6|        |      |    main:test_all2all.c:line.70
||=========================
```



Heap statistics analysis (activated by –g heap)

```
Notes for table 5:

  Table option:
    -O heap_hiwater
  Options implied by table option:
    -d am@,ub,ta,ua,tf,nf,ac,ab -b pe=[mmm]

  This table shows only lines with Tracked Heap HiWater MBytes > 0.


Table 5:  Heap Stats during Main Program

 Tracked |  Total |  Total | Tracked | Tracked |PE[mmm]
    Heap | Allocs | Frees  | Objects |  MBytes |
 HiWater |        |        |     Not |     Not |
  MBytes |        |        |   Freed |   Freed |

   9.794 |    915 |   910 |       4 |   1.011 |Total
|-----------------------------------------------------
|    9.943 |   1170 |  1103 |      68 |   1.046 |pe.0
|    9.909 |    715 |   712 |       3 |   1.010 |pe.22
|    9.446 |   1278 |  1275 |       3 |   1.010 |pe.43
|=====================================================
```



Many more groups available

blas	Basic Linear Algebra subprograms
caf	Co-Array Fortran (Cray CCE compiler only)
hdf5	manages extremely large and complex data
heap	dynamic heap
io	includes stdio and sysio groups
lapack	Linear Algebra Package
math	ANSI math
mpi	MPI
omp	OpenMP API
omp-rtl	OpenMP runtime library
pthreads	POSIX threads
shmem	SHMEM
sysio	I/O system calls
system	system calls
upc	Unified Parallel C (Cray CCE compiler only)

Online information available

- **User guide**
  - http://docs.cray.com
- **Man pages**
- **To see list of reports that can be generated**

```
$ pat_report -O -h
```

- **Notes sections in text performance reports provide information and suggest further options**
- **Cray Apprentice² panel help**
- **pat_help**
  - interactive help on the Cray Performance toolset
  - FAQ available through pat_help

PGAS Labs and Examples

Roberto Ansaloni

PGAS Examples

- **Some codes are provided as programming examples**
- **Fortran Coarray: pdensity**
  - Compute density of primes
- **Fortran Coarray: rgather**
  - Compare MPI with 2 Coarrays implementations based on get or put
- **UPC: upc_ticks**
  - Shows usage of timing functions (Cray specific)
- **UPC: add**
  - Simple add code
- **UPC: affinity**
  - Checks threads affinity

PGAS Labs - Himeno

- Read the tutorial doc and implement the suggested modifications
- Profile the MPI code with Cray perftools
- As suggested introduce coarray buffers
- Or introduce coarrays in another way:
  - Rewriting the code with direct get/put on pressure array
  - Introducing coarrays with the minimum number of modifications
- Profile the CAF code with Cray perftools
- Repeat the process porting the MPI C code to UPC