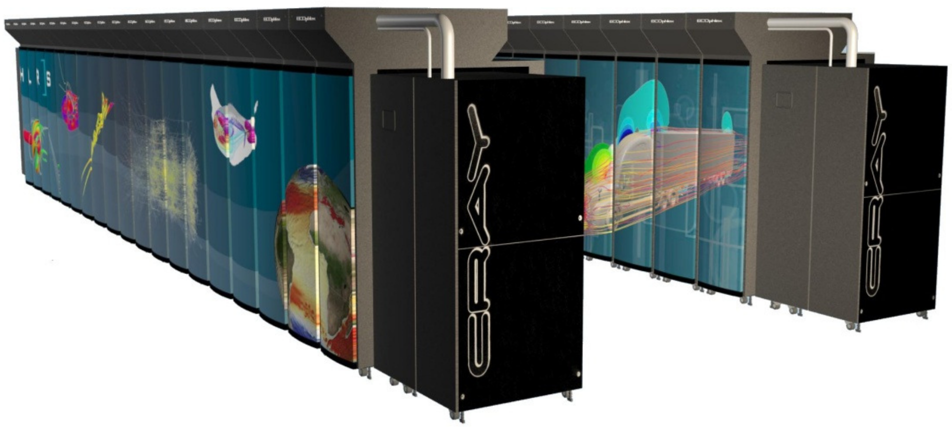


Applications of UPC

Jason J Beech-Brandt
Cray UK

THE SUPERCOMPUTER COMPANY

Outline

- UPC basics….
  - Shared and Private data – scalars and arrays
  - Pointers
  - Dynamic memory
- UPC in use
  - GUPS (Global Random Access) benchmark
- Cray support for PGAS
  - Legacy – X1, X1E, T3E
  - Current – X2, XT, XE

Context

- Most parallel programs are written using either:
  - Message passing with a SPMD model
    - Usually for scientific applications with C/Fortran
    - Scales easily
  - Shared memory with threads in OpenMP, Threads+C/C++/Fortran or Java
    - Usually for non-scientific applications
    - Easier to program, but less scalable performance
- Global Address Space (GAS) Languages take the best of both
  - global address space like threads (programmability)
  - SPMD parallelism like MPI (performance)
  - local/global distinction, i.e., layout matters (performance)

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
- Multiple PGAS languages: UPC (C), CAF (Fortran), Titanium (Java)

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);
}
```

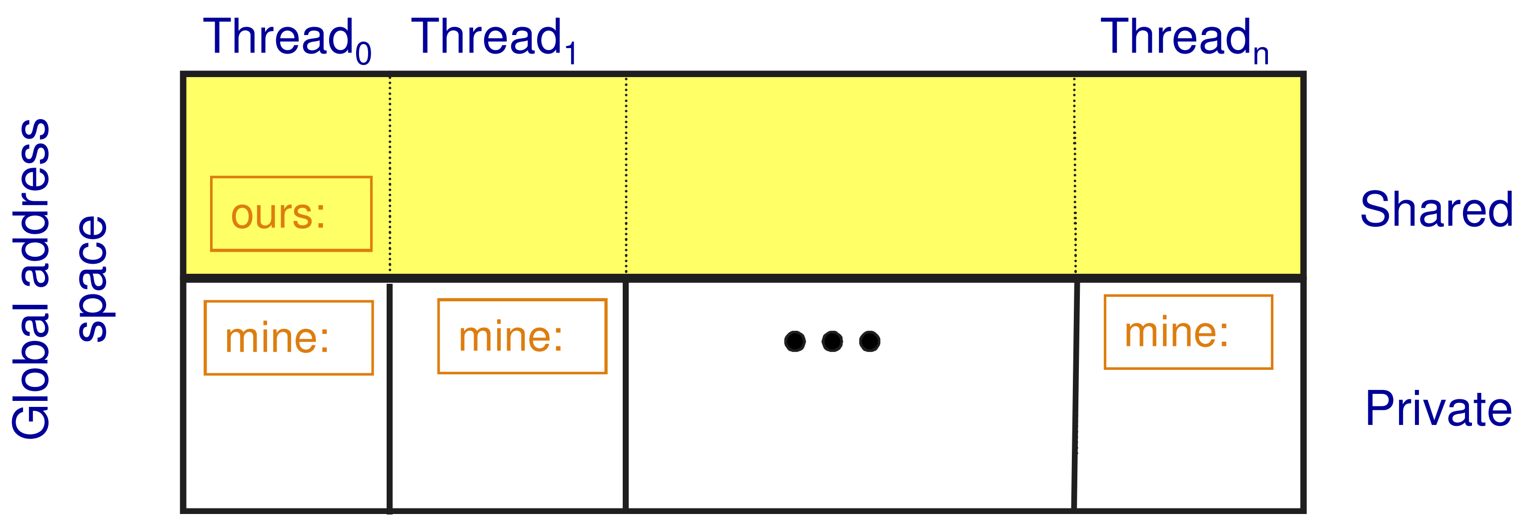
CRAY THE SUPERCOMPUTER COMPANY

HLRS

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;
```



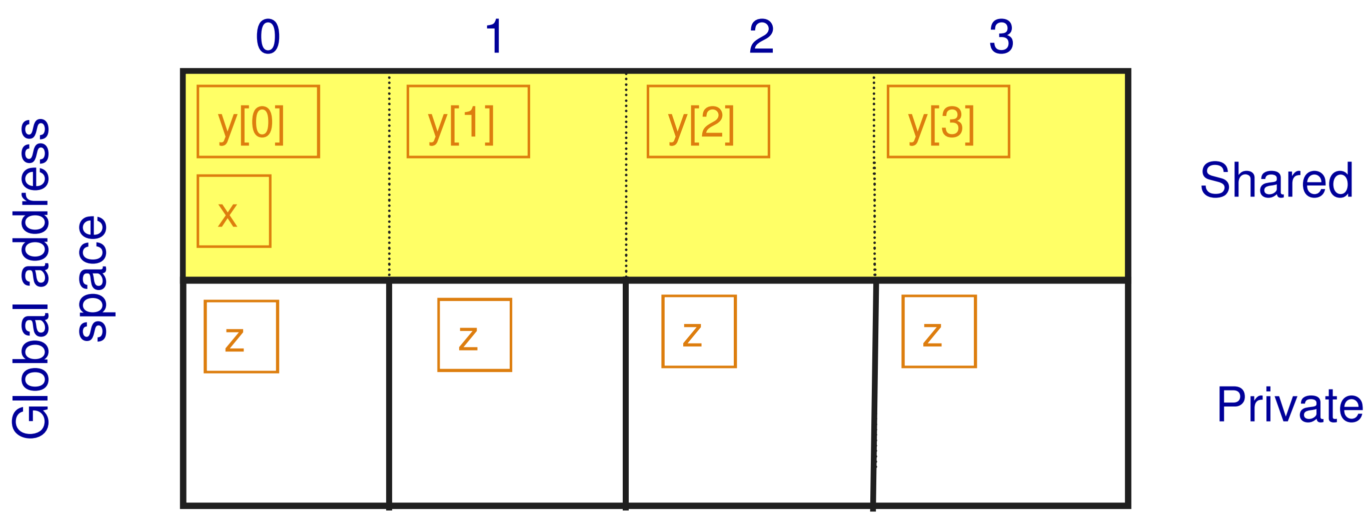
Shared and Private Data

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;
```

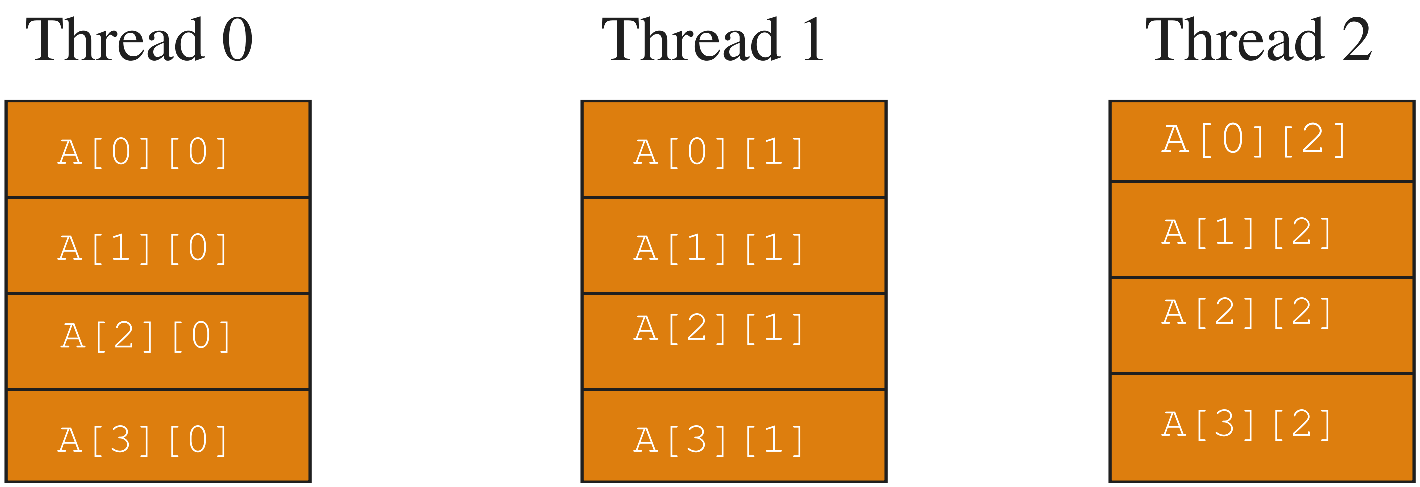
will result in the layout:



Shared and Private Data

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

will result in the following data layout:



THE SUPERCOMPUTER COMPANY

Blocking of Shared Arrays

- 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];`
  - e.g.: `shared [4] int a[16];`

Blocking of Shared Arrays

- 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

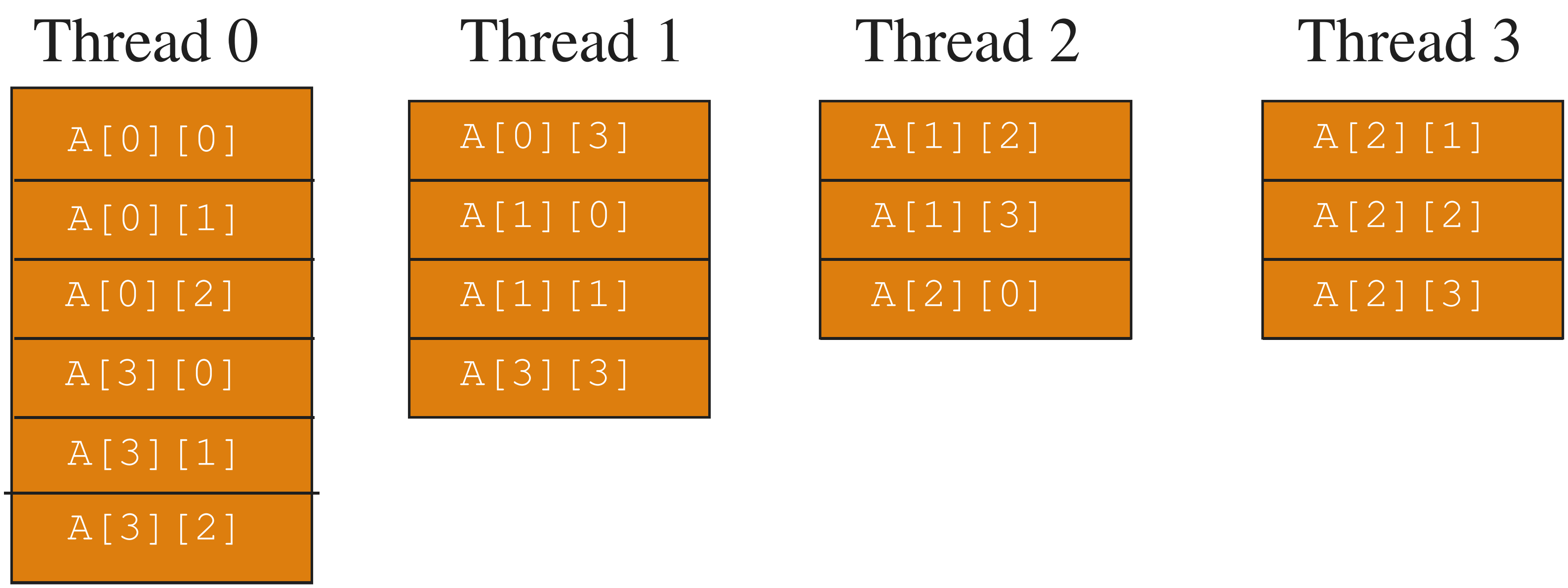
- Shared objects placed in memory based on affinity
- Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer
- All non-array shared qualified objects, i.e. shared scalars, have affinity to thread 0
- Threads access shared and private data

Shared and Private Data

Assume THREADS = 4

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

will result in the following data layout:



upc_forall

- A vector addition can be written as follows…
  - The code would be correct but slow if the affinity expression were `i+1` rather than `i`.

```
#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];
}
```

The cyclic data distribution may perform poorly on some machines

UPC Pointers

Where does the pointer point?

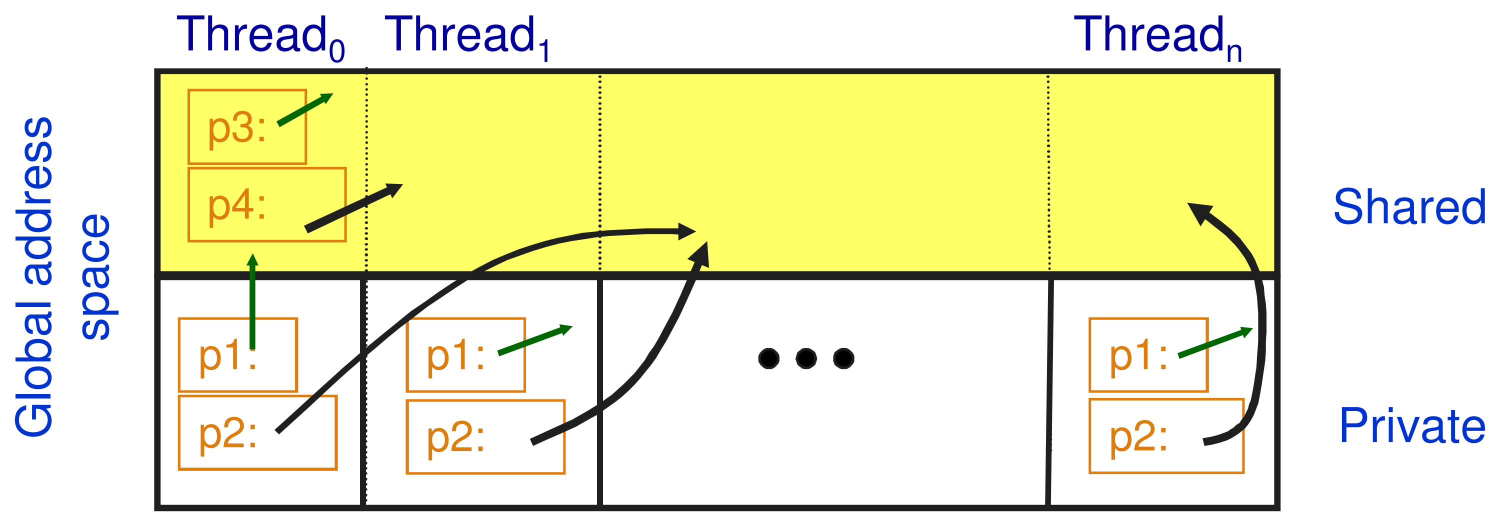
Where does the pointer reside?

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

```
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 */
```

Shared to private is not recommended.

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 */
```

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.

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

UPC Pointers

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)

Phase	Thread	Virtual Address

63 48 37

Example: Cray T3E implementation

UPC Pointers

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa !
- When casting a pointer-to-shared to a pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast

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"

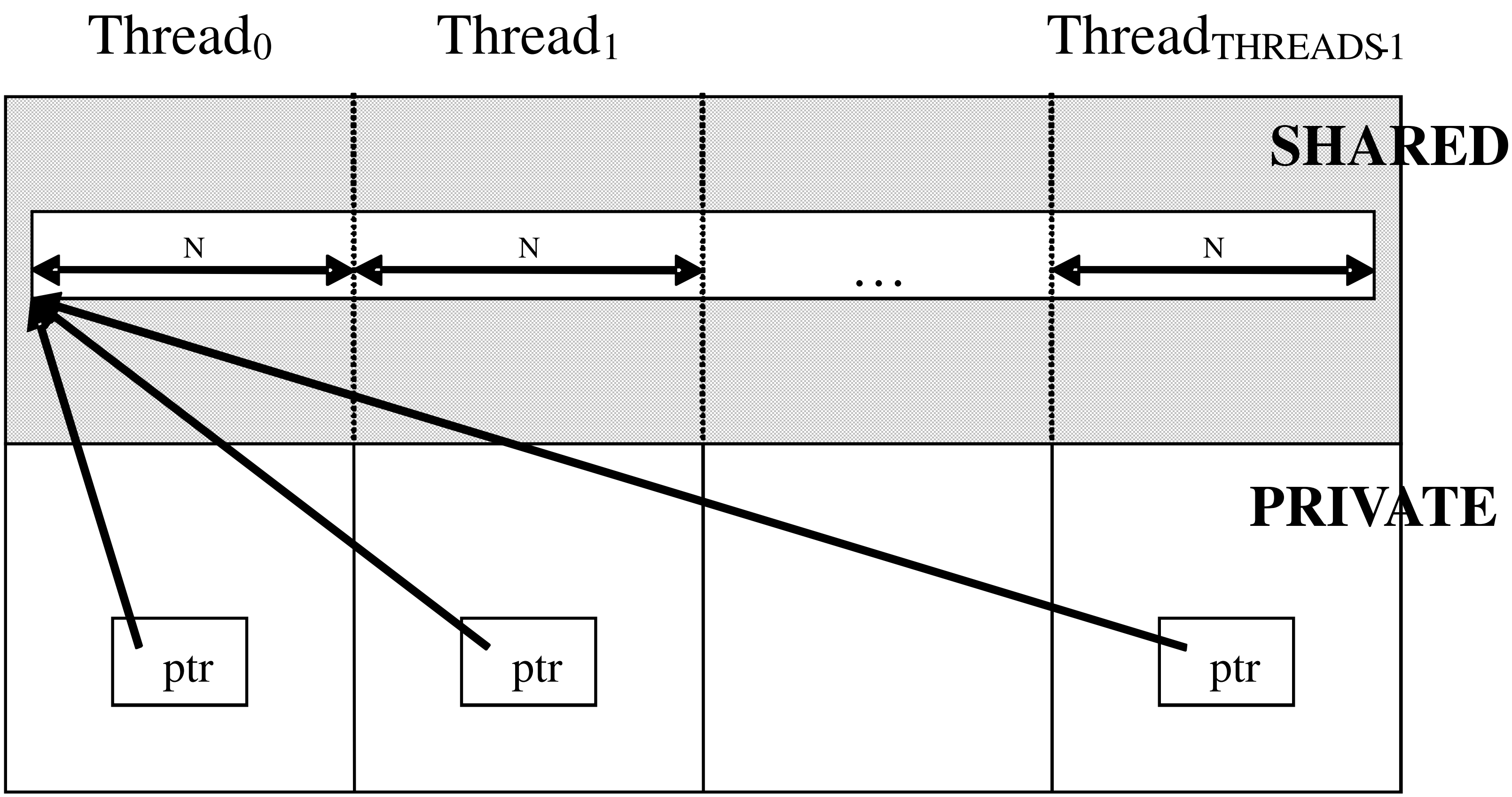
Collective Global Memory Allocation

```
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 ) );
```

CRAY THE SUPERCOMPUTER COMPANY

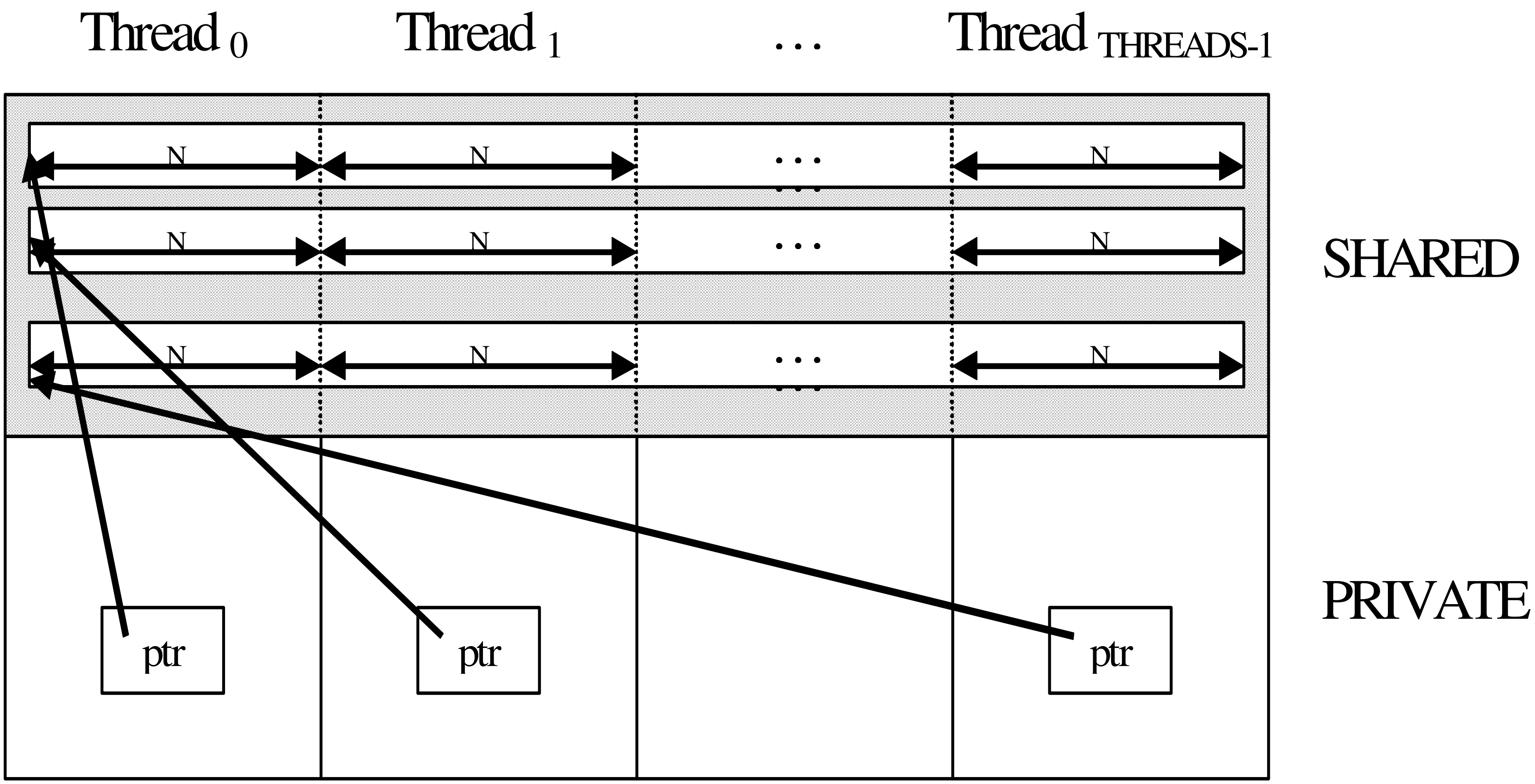
HLRS

Global Memory Allocation

```
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



Thread 0
Thread 1
...
Thread THREADS-1
N
N
...
N
SHARED
PRIVATE
ptr
ptr
ptr

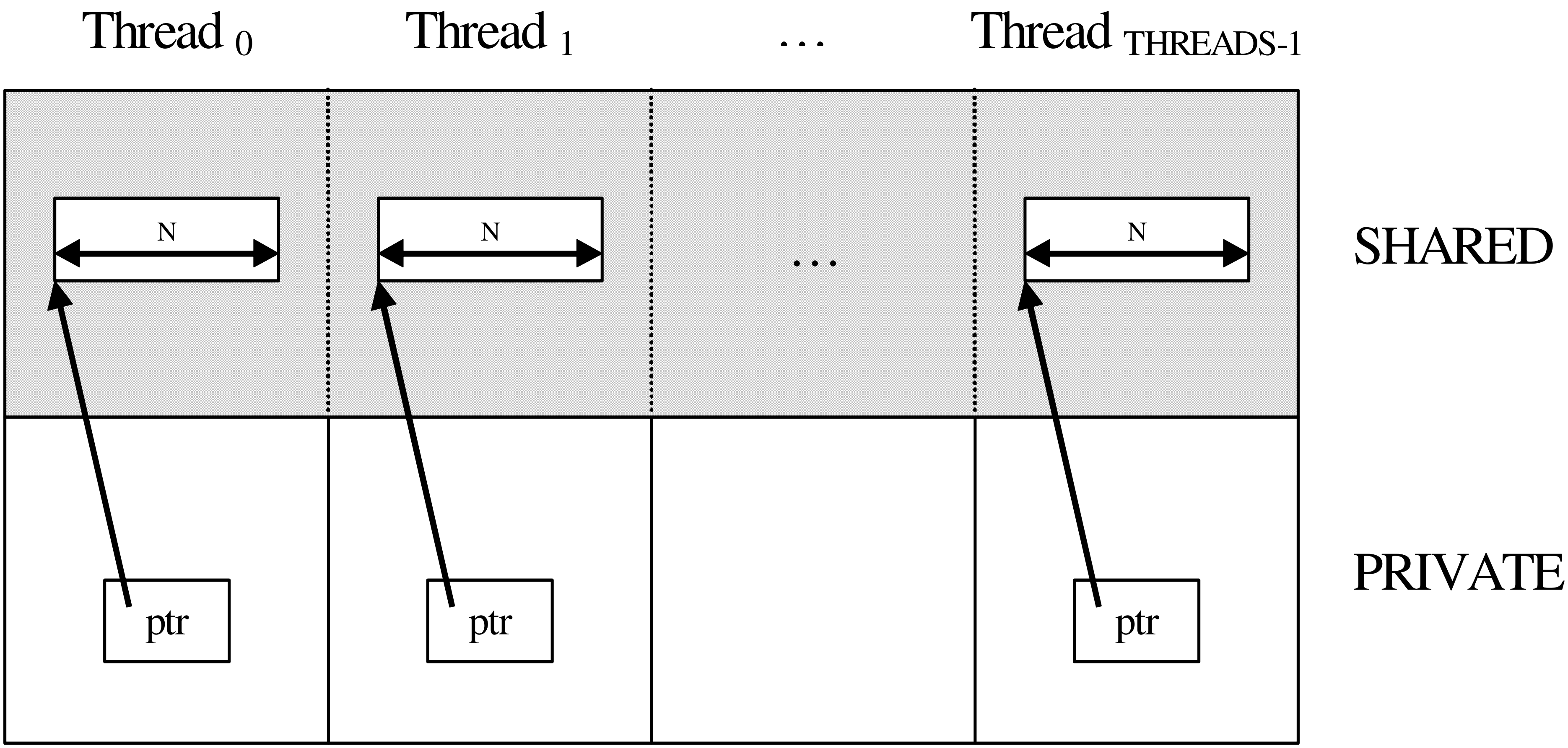
Local-Shared Memory Allocation

```
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

- `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

Lots more I haven't mentioned!

- 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

UPC Random Access: Designed for Speed

- This version of UPC Random Access was originally written by Nathan Wichmann in Spring 2004
- Written to maximize speed
- Had to work inside of the HPCC benchmark
- Had to run well on any number of CPUs
- Also happens to be a very productive way of writing the Global RA.

UPC Random Access: Highlights

- Trivial to parallelize, each PE gets its share of updates
- Unified Parallel C allows direct, one-sided access to distributed variables; NO two-sided messages!
- Decomposed “Table” into 2 Dims. to allow explicit, fast computation of LocalOffset and PE number
- Serial version is very succinct….

```
u64Int Ran;
Ran = 1;
for (i=0; i<NUPDATE; i++) {
   Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ?POLY : 0);
   GlobalOffset = Ran & (TABSIZE -1);
   Table[GlobalOffset] ^= Ran;
}
```

Productivity: Fewer lines of code

UPC VERSION

```
#pragma _CRI concurrent
for (j=0; j<STRIPSIZE; j++)
 for (i=0; i<SendCnt/STRIPSIZE; i++) {
  VRan[j] = (VRan[j] << 1) ^ ((s64Int) VRan[j]<
    ZERO64B ? POLY : ZERO64B);
  GlobalOffset = VRan[j] & (TableSize - 1);
  if (PowerofTwo)
    LocalOffset=GlobalOffset>>logNumProcs ;
  else
    LocalOffset=(double)GlobalOffset/(double)TH
    READS;
    WhichPe=GlobalOffset-LocalOffset*THREADS;
    Table[LocalOffset][WhichPe] ^= VRan[j] ;
 }
}
```

BASE VERSION

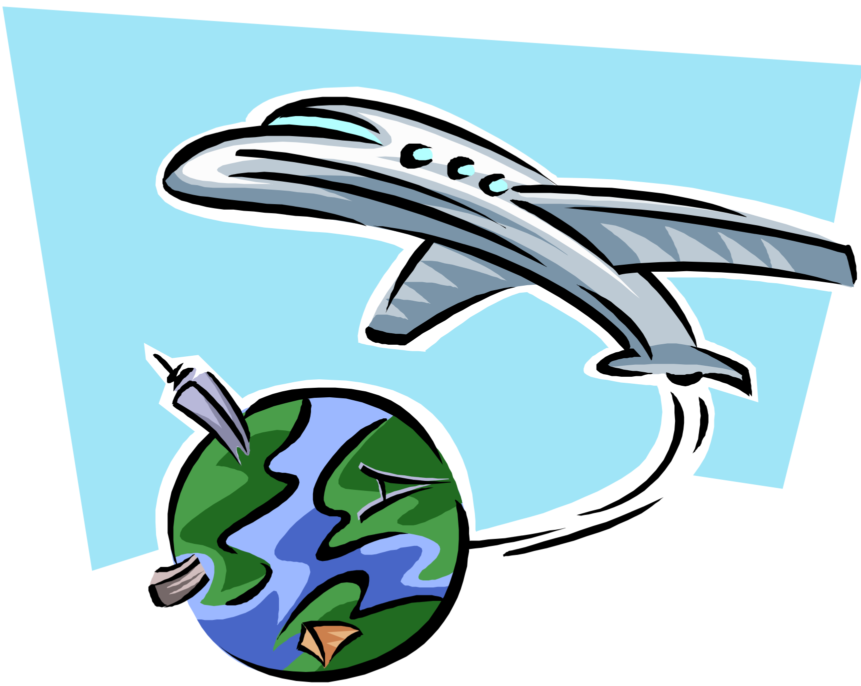
```
NumRecvs = (NumProcs > 4) ?(Mmin(4,MAX_RECV)) :
    1;
  for (j = 0; j < NumRecvs; j++)
    MPI_Irecv(&LocalRecvBuffer[j*LOCAL_BUFFER_SIZ
    E], localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD,&inreq[j]);
while (i < SendCnt) {
do {
MPI_Testany(NumRecvs, inreq, &index, &have_done,
    &status);
if (have_done) {
 if (status.MPI_TAG == UPDATE_TAG) {
   MPI_Get_count(&status, INT64_DT,
    &recvUpdates);
bufferBase = index*LOCAL_BUFFER_SIZE;
for (j=0; j < recvUpdates; j ++) {
 inmsg = LocalRecvBuffer[bufferBase+j];
 LocalOffset = (inmsg & (TableSize - 1)) -
    GlobalStartMyProc;
 HPCC_Table[LocalOffset] ^= inmsg;
 }
 } else if (status.MPI_TAG == FINISHED_TAG) {
    NumberReceiving--;
 } else {
    abort();
 }
```

CRAY THE SUPERCOMPUTER COMPANY

H L R S

Productivity : Fewer lines of code

UPC VERSION

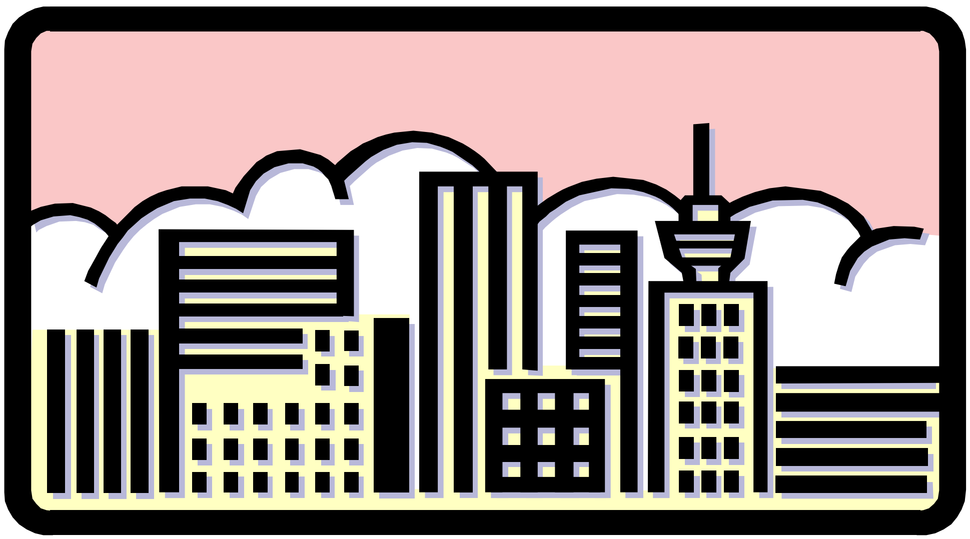


BASE VERSION

```
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZE],
     localBufferSize,INT64_DT, MPI_ANY_SOURCE,
     MPI_ANY_TAG, MPI_COMM_WORLD,&inreq[index]);
}
} while (have_done && NumberReceiving > 0);
  if (pendingUpdates < maxPendingUpdates) {
    Ran = (Ran << 1) ^ ((s64Int) Ran <   ZERO64B ?
    POLY : ZERO64B);
    GlobalOffset = Ran & (TableSize-1);
    if ( GlobalOffset < Top)
      WhichPe = ( GlobalOffset / (MinLocalTableSize
    + 1) );
   else
    WhichPe = ( (GlobalOffset - Remainder) /
    MinLocalTableSize );
   if (WhichPe == MyProc) {
    LocalOffset = (Ran & (TableSize - 1)) -
    GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= Ran;
   }
   else {
    HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
   }
   i++;
 }
 else {
```

Productivity : Fewer lines of code

UPC VERSION



BASE VERSION

```
MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
   if (have_done) {
    outreq = MPI_REQUEST_NULL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer,
     localBufferSize, &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, INT64_DT,
     (int)pe, UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
   }}}
while (pendingUpdates > 0) {
do {
MPI_Testany(NumRecvs, inreq, &index, &have_done,
     &status);
if (have_done) {
  if (status.MPI_TAG == UPDATE_TAG) {
   MPI_Get_count(&status, INT64_DT, &recvUpdates);
   bufferBase = index*LOCAL_BUFFER_SIZE;
  for (j=0; j < recvUpdates; j ++) {
   inmsg = LocalRecvBuffer[bufferBase+j];
   LocalOffset = (inmsg & (TableSize - 1)) -
     GlobalStartMyProc;
   HPCC_Table[LocalOffset] ^= inmsg;
   }
} else if (status.MPI_TAG == FINISHED_TAG) {
  NumberReceiving--;
```

Productivity : Fewer lines of code

UPC VERSION

BASE VERSION

```
} else {
  abort();}
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZE],
    localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD,&inreq[index]);
}} while (have_done && NumberReceiving > 0);
    MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
    if (have_done) {
      outreq = MPI_REQUEST_NULL;
      pe = HPCC_GetUpdates(Buckets,
    LocalSendBuffer, localBufferSize, &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, INT64_DT,
    (int)pe, UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
    } }
for (proc_count = 0 ; proc_count < NumProcs ;
    ++proc_count) {
  if (proc_count == MyProc) { finish_req[MyProc] =
    MPI_REQUEST_NULL; continue; }
  MPI_Isend(&Ran, 1, INT64_DT, proc_count,
    FINISHED_TAG,MPI_COMM_WORLD, finish_req +
    proc_count);
  }
while (NumberReceiving > 0) {
```

Productivity : Fewer lines of code

UPC VERSION

BASE VERSION

```
MPI_Waitany(NumRecvs, inreq, &index, &status);
if (status.MPI_TAG == UPDATE_TAG) {
  MPI_Get_count(&status, INT64_DT, &recvUpdates);
  bufferBase = index * LOCAL_BUFFER_SIZE;
for (j=0; j < recvUpdates; j ++) {
  inmsg = LocalRecvBuffer[bufferBase+j];
  LocalOffset = (inmsg & (TableSize - 1)) -
    GlobalStartMyProc;
  HPCC_Table[LocalOffset] ^= inmsg;
  }
  } else if (status.MPI_TAG == FINISHED_TAG){
     NumberReceiving--;
    } else {
      abort(); }
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZ
    E], localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, &inreq[index]);
}
MPI_Waitall( NumProcs, finish_req,
    finish_statuses);
HPCC_FreeBuckets(Buckets, NumProcs);
for (j = 0; j < NumRecvs; j++) {
    MPI_Cancel(&inreq[j]);
    MPI_Wait(&inreq[j], &ignoredStatus);
  }
```

CRAY THE SUPERCOMPUTER COMPANY

HLRS

Productivity: Algorithm Transparency

Generate Random Number

Compute GO

Decompose GO into LO and WhichPE

XOR VRan and Table

```
#pragma _CRI concurrent
for (j=0; j<STRIPSIZE; j++)
 for (i=0; i<SendCnt/STRIPSIZE; i++) {
  VRan[j] = (VRan[j] << 1) ^ ((s64Int)VRan[j]
            < ZERO64B ? POLY : ZERO64B);
  GlobalOffset = VRan[j] & (TableSize - 1);

  if (PowerofTwo)
    LocalOffset=GlobalOffset>>logNumProcs ;
  else
    LocalOffset=
       (double)GlobalOffset/(double)THREADS;
  WhichPe=GlobalOffset-LocalOffset*THREADS;

  Table[LocalOffset][WhichPe] ^= VRan[j] ;
}}
```

CRAY THE SUPERCOMPUTER COMPANY

H L R S

Productivity + Speed = Results

- UPC Random Access sustains 7.69 GUPs on 1008 Cray X1E MSPs.
- Works inside the HPCC framework
- Is "in the spirit" of the benchmark
- Easy to understand and modify if computations are more complex

- The Future
  - Atomic XORs will vastly improve performance
    - All memory references will be "Fire and Forget"

PGAS and Cray

- Cray have been supporting CAF and UPC since the beginning
  - Original support on the T3E
- Full PGAS support on the Cray XT and XE
  - Cray Compiling Environment 7.0 – Dec 08
  - Cray Compiler Environment 7.3 – Dec 10
  - Full UPC 1.2 specification
  - Full CAF support – CAF proposed for the Fortran 2008 standard
  - Hybrid MPI/PGAS codes supported – very important!
- Fully integrated with the Cray software stack
  - Same compiler drivers, job launch tools, libraries
  - Integrated with Craypat – Cray performance tools
- Hardware support for PGAS in Gemini interconnect

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