Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

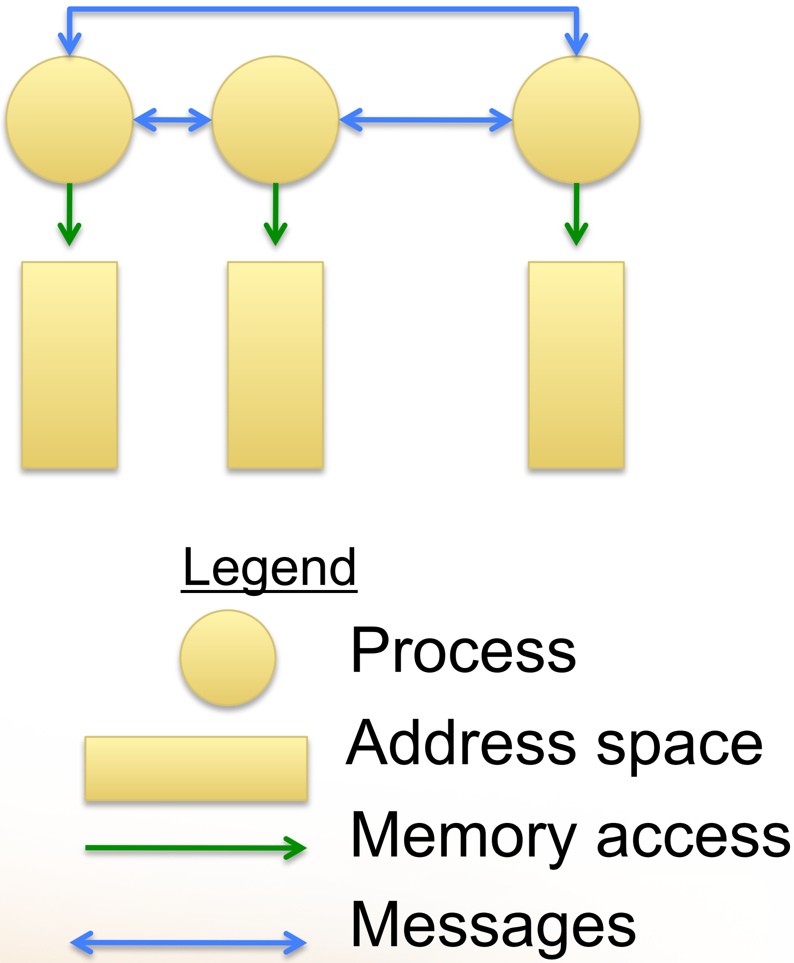
Programming Models

- What is a programming model?
  - The logical interface between architecture and applications
- Why programming models?
  - Decouple applications and architectures
    - Write applications that run effectively across architectures
    - Design new architectures that can effectively support legacy applications
- Programming model design considerations
  - Expose modern architectural features to exploit machine power and improve performance
  - Maintain ease of use

Examples of Parallel Programming Models

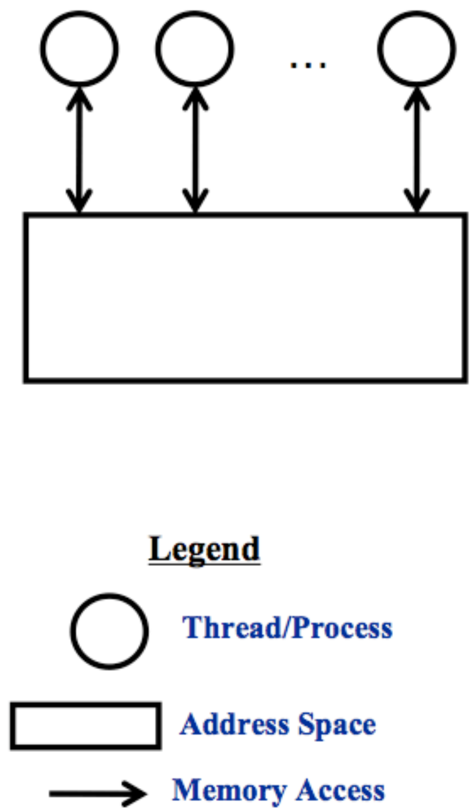
- **Message Passing**
- **Shared Memory (Global Address Space)**
- **Partitioned Global Address Space (PGAS)**

The Message Passing Model



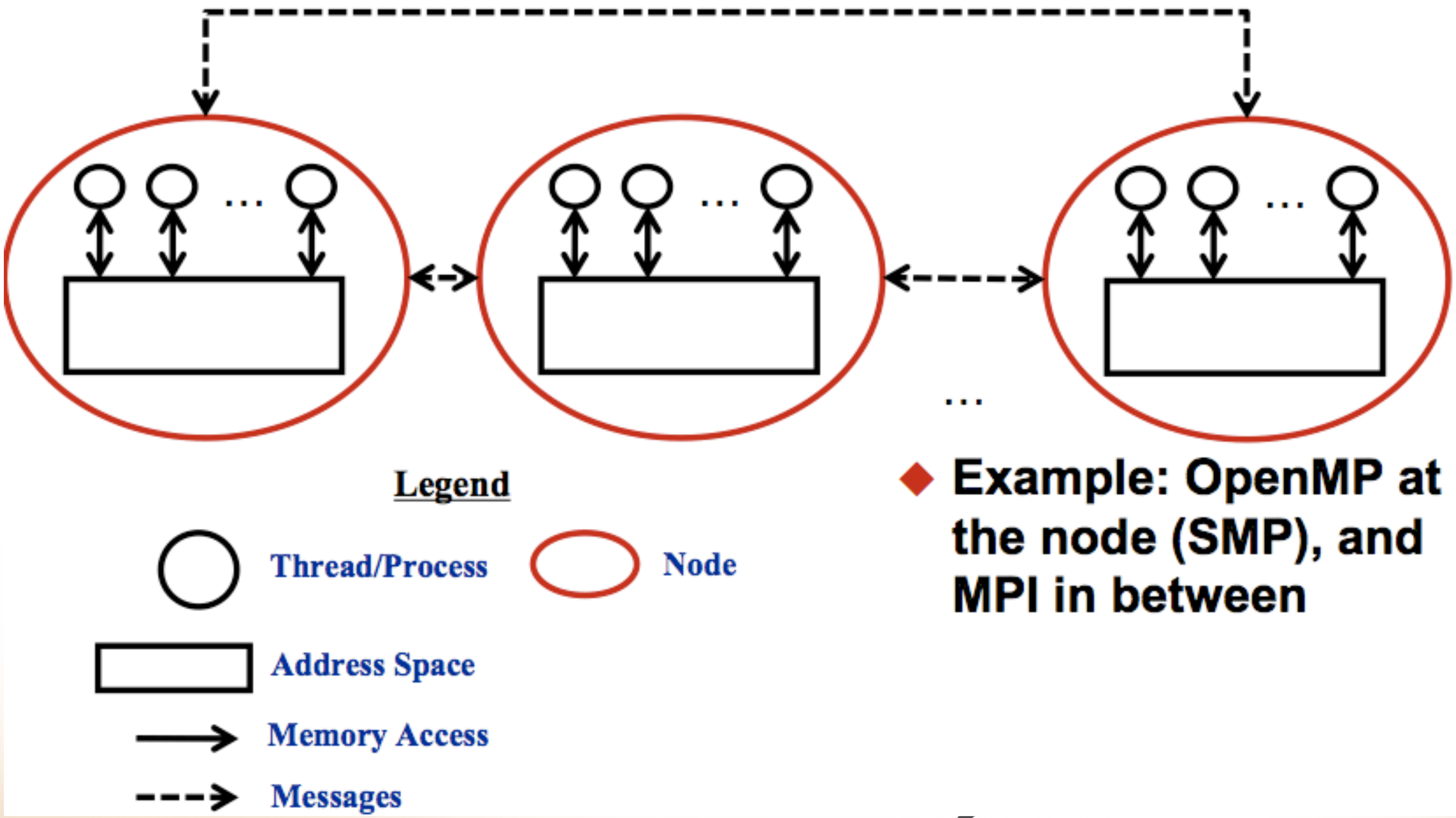
- Concurrent sequential processes
- Explicit communication
- Library-based
- Pros:
  - Programmer controls data and work distribution
- Cons:
  - Significant communication overhead for small transactions
  - Excessive buffering
  - Hard to program
- Example: MPI

The Shared Memory Model



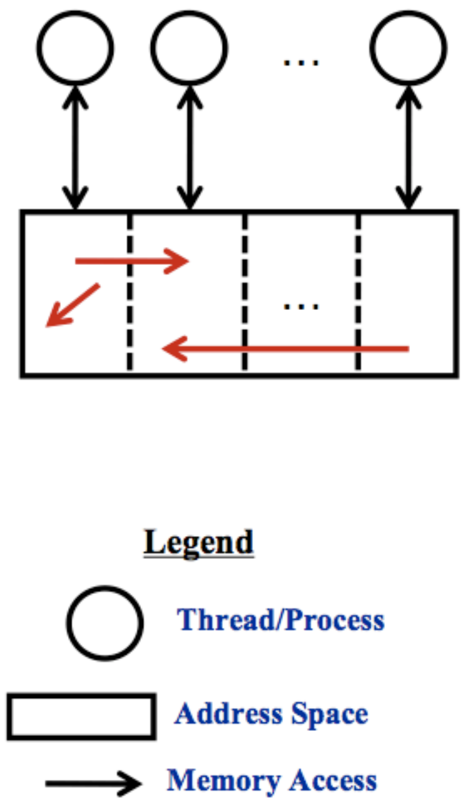
- Concurrent threads with shared space
- Positive:
  - Simple statements
  - Read remote memory via an expression
  - Write remote memory through assignment
- Negative:
  - Manipulating shared data leads to synchronization requirements
  - Does not allow locality exploitation
- Example: OpenMP, Java

Hybrid Model
Example: Message Passing + Shared Memory



- Example: OpenMP at the node (SMP), and MPI in between

The PGAS Model



- Concurrent threads with a partitioned shared space
  - A datum may reference data in other partitions
  - Global arrays have fragments in multiple partitions
- Positive:
  - Helps in exploiting locality
  - Simple statements as shared memory
- Negative:
  - sharing all memory can result in subtle bugs and race conditions
  - Examples: UPC, X10, Chapel, CAF, Titanium

PGAS Model (2)

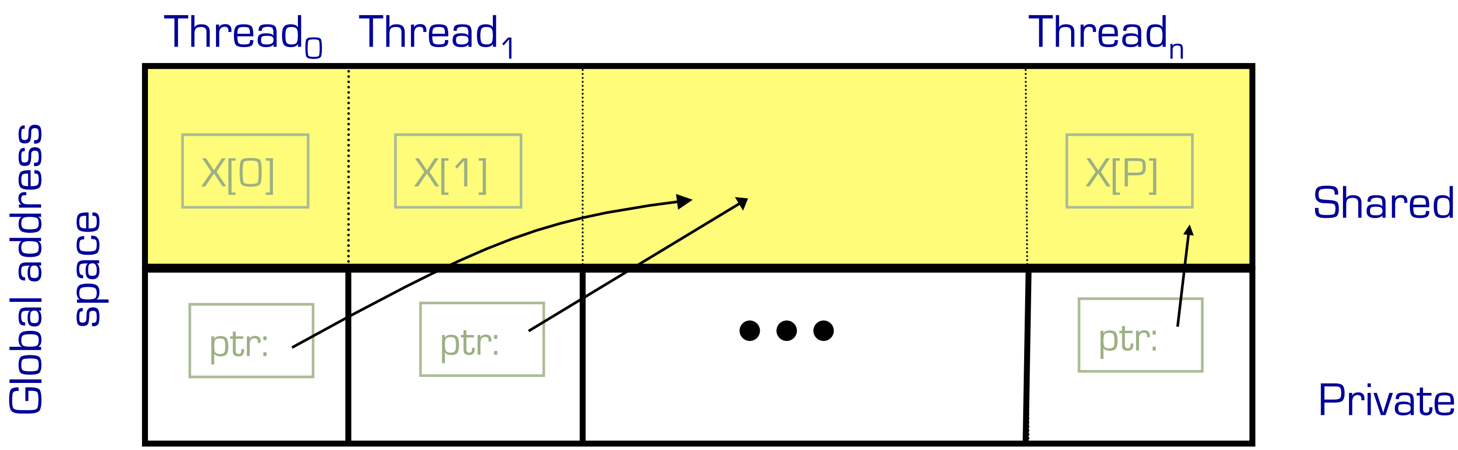
A collection of threads operating in a partitioned global address space that is logically distributed among threads. Each thread has affinity with a portion of the globally shared address space. Each thread has also a private space.

Elements in partitioned global space belonging to a thread are said to have affinity to that thread.

PGAS vs. Other Programming Models

	UPC, X10, Chapel, CAF, Titanium	MPI	OpenMP
Memory model	PGAS	Distributed Memory	Shared Memory
Notation	Language	Library	Annotations
Global arrays?	Yes	No	No
Global pointers/ references?	Yes	No	No
Locality exploitation?	Yes	Yes, necessarily	No

Global Address Space Eases Programming



- The languages share the global address space abstraction
  - Shared memory is logically partitioned by processors
  - Remote memory may stay remote: no automatic caching implied
  - One-sided communication: reads/writes of shared variables
  - Both individual and bulk memory copies
- Languages differ on details
  - Some models have a separate private memory area
  - Distributed array generation and how they are constructed

Current Implementations of PGAS Languages

- A successful language/library must run everywhere
- UPC
  - **Commercial compilers available on Cray, IBM, SGI, HP machines**
  - **Open source compiler from LBNL/UCB (source-to-source)**
  - **Open source gcc-based compiler from Intrepid**
- CAF
  - **Commercial compiler available on Cray machines**
  - **Open source compiler available from Rice**
- Titanium
  - **Open source compiler from UCB runs on most machines**
- Common tools
  - **Open64 open source research compiler infrastructure**
  - **ARMCI, GASNet for distributed memory implementations**
  - **Pthreads, System V shared memory**

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

What is UPC?

- UPC - Unified Parallel C
  - An explicitly parallel extension of ANSI C
  - A distributed shared memory parallel programming language
  - Enables programmers to exploit data locality on a variety of memory architectures
- 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 into trouble.
  - Common and familiar C syntax and semantics with simple extensions for thread parallelism with shared data

ICL UT

UPC Specifications

- UPC consortium of government, academia, HPC vendors, including:
  - ARSC, Compaq, CSC, Cray Inc., Etnus, GWU, HP, IBM, IDA CSC, Intrepid Technologies, LBNL, LLNL, MTU, NSA, UCB, UMCP, UF, US DOD, US DOE, OSU
- Set of specs for a parallel C
  - v1.0 completed February of 2001
  - v1.1.1 in October of 2003
  - v1.2 in May of 2005
- See http://upc.gwu.edu for more detail
- UPC: Distributed Shared Memory Programming; Authors: Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick; ISBN: 0-471-22048-5 ; Published by John Wiley and Sons- May, 2005

UPC Implementations

- Many UPC implementations are available
  - Cray CLE
  - IBM XL UPC
  - GCC UPC
  - HP UPC
  - SGI UPC
  - Berkeley UPC Compiler

Example 1 : Hello World

```
#include <upc_relaxed.h>
#include <stdio.h>
void main() {
    printf("Hello World from THREAD %d (of %d THREADS)\n",
    MYTHREAD, THREADS);
}
```

- The keyword THREADS signifies the number of threads that the current execution is utilizing.
  - The value of THREADS can be defined either at compile time or at runtime.
- The keyword MYTHREAD is used to determine the thread number currently being executed.

Sequential vector addition

```
//vect_add.c

#define N 1000
int v1[N], v2[N], v1plusv2[N];
void main()
{
   int i;
   for (i=0; i<N; i++)
    v1plusv2[i]=v1[i]+v2[i];
}
```

Example 2: parallel vector addition

```
//vect_add.c
#include <upc_relaxed.h>
#define N 1000
shared int v1[N], v2[N], v1plusv2[N];
void main()
{
  int i;
  upc_forall (i=0; i<N; i++; i)
   v1plusv2[i]=v1[i]+v2[i];
}
```

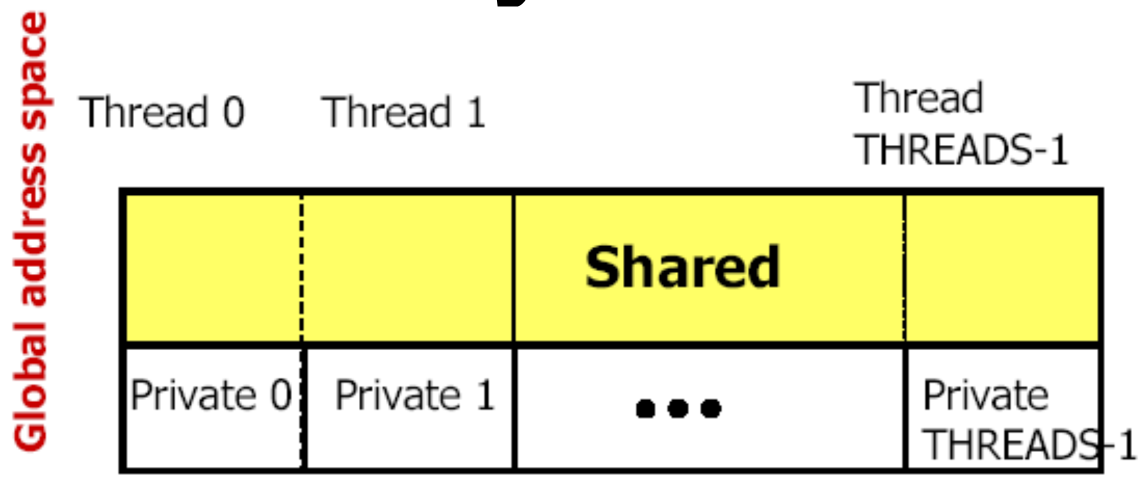
Parallel vector addition (2)

- In line 1 the inclusion of upc_relaxed.h signifies that this code will not follow the strict memory consistency model and will allow the compiler to optimize the order of shared accesses for the best performance.
- In line 3, the shared qualifier signifies that the variables will be shared among the threads, and since there is no block_size specified they will be distributed in a round robin manner across the threads until all data elements are exhausted**.**
- The upc_forall statement in line 7 specifies work sharing among the threads. The difference between a normal C for loop and the upc_forall loop is the fourth field, called the *affinity* field. The affinity field determines which thread will execute which iteration of the loop body. In this example, iteration i will be executed by thread i%THREADS. Given the round robin default distribution of the elements of the arrays, all computations in this example will be local and require no remote memory accesses.

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

UPC memory model



- A pointer-to-shared can reference all locations in the shared space
- A pointer-to-local ("plain old C pointer") may only reference addresses in its private space or addresses in its portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory

UPC execution model

- A number of threads working independently in SPMD fashion
  - Similar to MPI
  - MYTHREAD specifies thread index (0..THREADS-1)
  - Number of threads specified at compile-time or run-time
- Synchronization only when needed
  - Barriers
  - Locks
  - Memory consistency control

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Shared scalar and array data

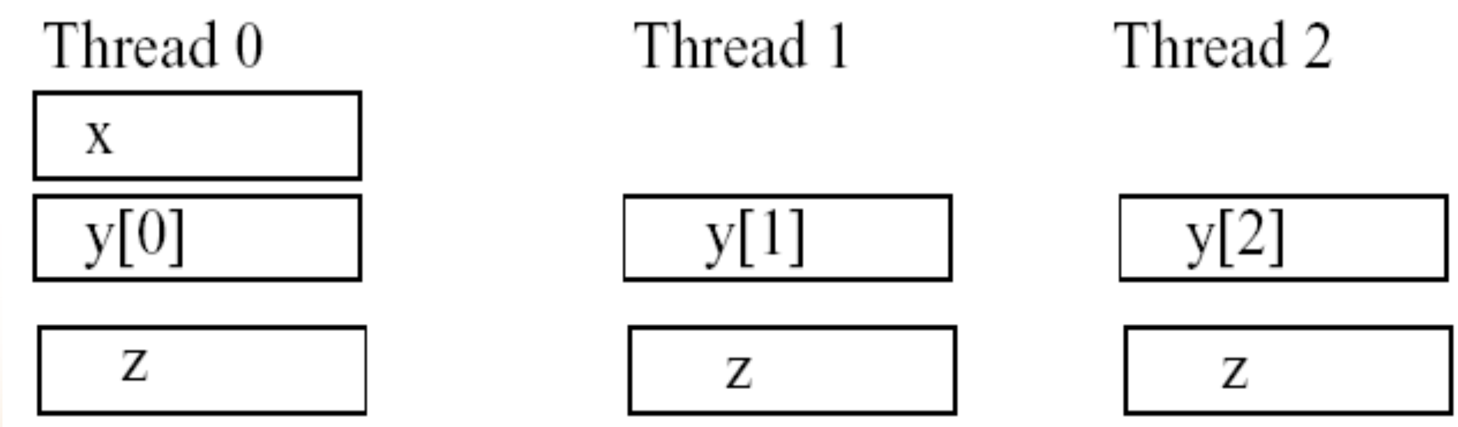
- Shared array elements and blocks can be spread across the threads
  - `shared int x[THREADS]`
    `/* One element per thread */`
  - `shared int y[10][THREADS]`
    `/* 10 elements per thread */`
- Scalar data declarations
  - `shared int a;`
    `/* One item in global space (affinity to thread 0) */`
  - `int b;`
    `/* one private b at each thread */`

Shared and private data

- Example (assume THREADS = 3):

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

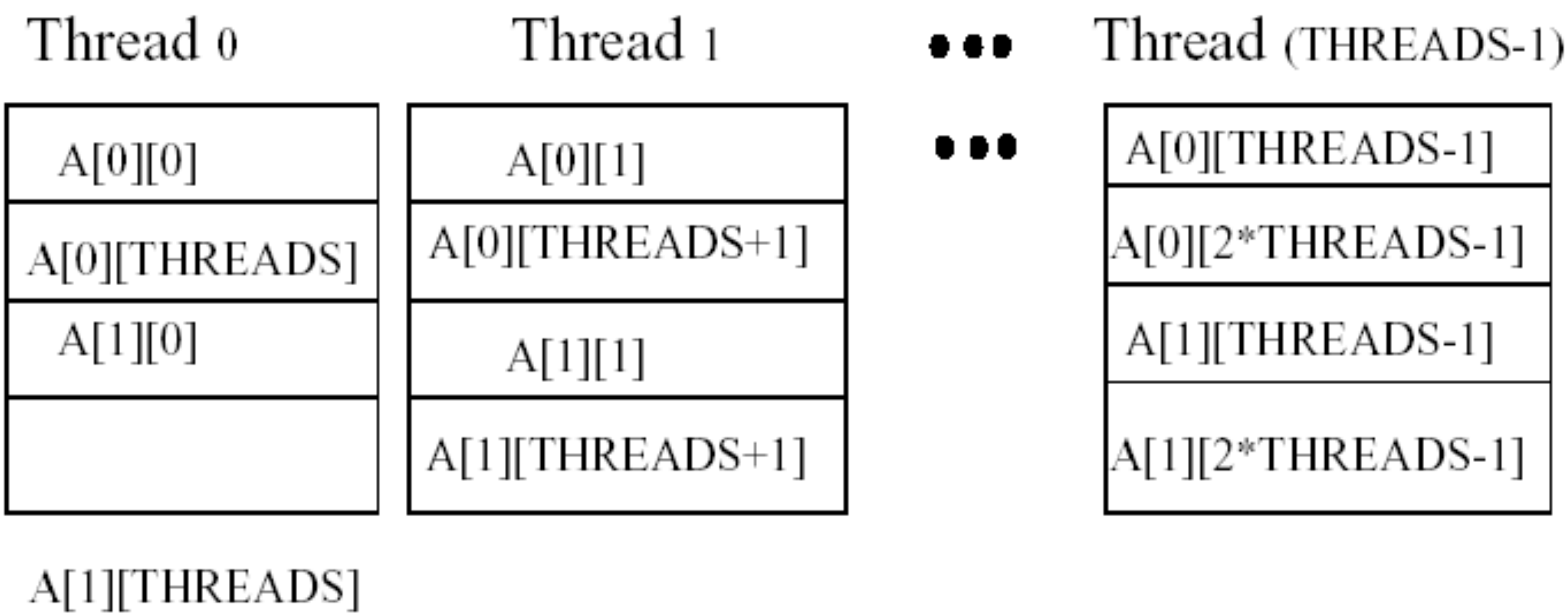
- The resulting layout is:



Shared data

shared int A[2][2*THREADS];

will result in the following data layout:



Remember: C uses row-major ordering

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

Blocking of shared arrays (2)

- 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:

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

Blocking of shared arrays (3)

- Assuming THREADS = 4

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

  will result in the following data layout:

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]			

Data distributions for shared arrays

- UPC official spec only supports 1d block cyclic
- IBM xlupc compiler supports more general data distribution : 'multi-dimensional blocking'
- Eg : `shared [2][2] double A[5][5];`
- Divide the array into multidimensional tiles
- Distribute the tiles among processors in cyclic fashion
- More general than UPC spec, but not as general as ScaLAPACK or HPF

Multidimensional Blocking

```
shared [2][2] double A[5][5];
```

0	0	1	1	2
0	0	1	1	2
3	3	0	0	1
3	3	0	0	1
2	2	3	3	0

2D Array Layouts in UPC

- Array a1 has a row layout and array a2 has a block row layout.

```
shared [m] int a1 [n][m];
shared [k*m] int a2 [n][m];
```

- To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.
  - Assume r*c = THREADS;

    shared [b1][b2] int a5 [m][n][r][c][b1][b2];
  - or equivalently

    shared [b1*b2] int a5 [m][n][r][c][b1][b2];
- Can use arrays of pointers for more general data distributions

Shared and private data - summary

- 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 scalar shared qualified objects have affinity with thread 0.
- Threads may access shared and private data.

UPC Pointers

Where does the pointer point?

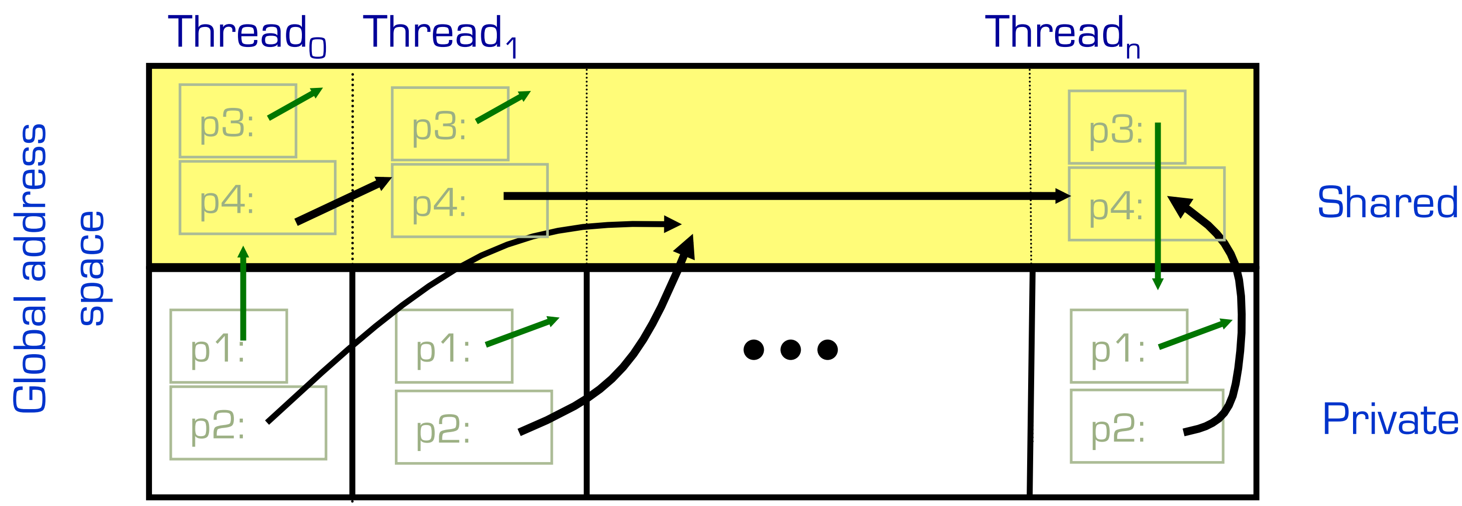
Where does the pointer reside?

	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 (2)




```
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

- 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.

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Dynamic memory allocation

- 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.

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 space in the shared space.
- If called by more than one thread, multiple regions are allocated and each thread which makes the call gets a different pointer.
- Space allocated per calling thread is equivalent to :

shared [nbytes] char[nblocks * nbytes]

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]

Freeing memory

```
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.

Some memory functions in UPC

- Equivalent of memcpy :
  - `upc_memcpy(dst, src, size)`
    `/* copy from shared to shared */`
  - `upc_memput(dst, src, size)`
    `/* copy from private to shared */`
  - `upc_memget(dst, src, size)`
    `/* copy from shared to private */`
- Equivalent of memset:
  - `upc_memset(dst, char, size)`
    `/* initialize shared memory with a character */`

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Work sharing with upc_forall()

- Distributes independent iterations
- Each thread gets a bunch of iterations.
- Affinity (expression) field determines how to distribute work.
- Simple C-like syntax and semantics

```
upc_forall (init; test; loop; expression)
  statement;
```

- Function of note:

```
upc_threadof(shared void *ptr)
```

returns the thread number that has affinity to the pointer-to-shared.

Synchronization

- No implicit synchronization among the threads
- UPC provides the following synchronization mechanisms:
  - Barriers
  - Locks
  - Fence
  - Spinlocks (using memory consistency model)

Synchronization: barriers

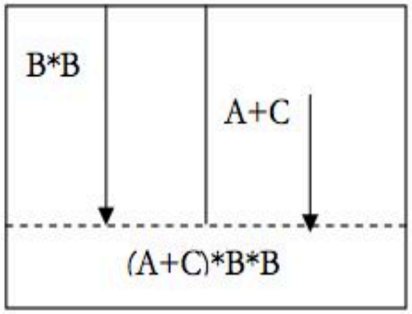
- UPC provides the following barrier synchronization constructs:
  - Barriers (Blocking)
    - upc_barrier {expr};
  - Split-Phase Barriers (Non-blocking)
    - upc_notify {expr};
    - upc_wait {expr};
    - Note: upc_notify is not blocking, upc_wait is blocking

Split-barrier example

```
shared [N]int A[N][N];
shared [N]int C[N][N];
shared [N]int B[N][N];
shared [N]int ACsum[N][N];
shared [N]int Bsqr[N][N];
shared [N]int Result[N][N];

void matrix_multiplication (shared[N] int result[N][N],
                            shared[N] int m1[N][N],
                            shared[N] int m2[N][N]){
    int i, j, l, sum;
    upc_forall(i=0;i<N;i++; &m1[i][0]){
        for(j=0;j<N;j++){
            sum=0;
                for(l=0;l<N;l++)
                sum+=m1[i][l]*m2[l][j];
            result [i][j]=sum;
        }
    }
}

matrix_multiplication(Bsqr,B,B);
upc_notify 1;
upc_forall(i=0;i<N;i++;&A[i][0]){
    for(j=0;j<N;j++)
        ACsum[i][j]+=A[i][j]+C[i][j];
}
upc_wait 1;
matrix_multiplication(Result, ACsum, Bsqr);
```



Synchronization: fence

- UPC provides a fence construct.
  - Equivalent to a null strict reference, and has the syntax
    - `upc_fence;`
  - Null strict reference:
    - `{static shared strict int x; x=x;}`
- Ensures that all shared references issued before the upc_fence are complete

Synchronization: locks

- In UPC, shared data can be protected against multiple writers :
  - `void upc_lock(upc_lock_t *l)`
  - `int upc_lock_attempt(upc_lock_t *l) // returns 1 on success and 0 on failure`
  - `void upc_unlock(upc_lock_t *l)`
- Locks can be allocated dynamically. Dynamically allocated locks can be freed.
- Dynamic locks are properly initialized and static locks need initialization.

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Memory consistency model

- Has to do with the ordering of shared operations
- Under the relaxed consistency model, the shared operations can be reordered by the compiler/runtime system.
- The strict consistency model enforces sequential ordering of shared operations (all threads see same order of writes to shared variables).

Memory consistency model (2)

- User specifies the memory model through:
  - declarations
  - pragmas for a particular statement or sequence of statements
  - use of barriers and global operations
- Consistency can be *strict* or *relaxed*
- Programmer responsible for using correct consistency model

Memory consistency model (3)

- Default behavior can be controlled by the programmer:
  - Use strict memory consistency
    - `#include<upc_strict.h>`
  - Use relaxed memory consistency
    - `#include<upc_relaxed.h>`
- Default behavior can be altered for a statement or a block of statements using
  - `#pragma upc strict`
  - `#pragma upc relaxed`
- Default behavior can be altered for a variable definition using:
  - Type qualifiers: *strict* & *relaxed*

Memory consistency example

```
strict shared int flag_ready = 0;
shared int result0, result1;
if (MYTHREAD==0) {
    result0 = expression1;
    flag_ready=1; //if not strict, it could be
                  // switched with the above statement
} else if (MYTHREAD==1) {
    while (!flag_ready); //Same note as above
    result1=expression2+result0;
}
```

- We could have used a barrier between the first and second statement in the if and the else code blocks.
  - Expensive!! Affects all operations at all threads
- We could have used a fence in the same places.
  - Affects shared references at all threads!
- The above is an example of point-to-point synchronization.

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Example: matrix multiplication

- Given two integer matrices A(NxP) and B(PxM), we want to compute C =A x B.
- Entries cij in C are computed by the formula:

$$c_{ij} = \sum_{l=1}^{p} a_{il} \times b_{lj}$$

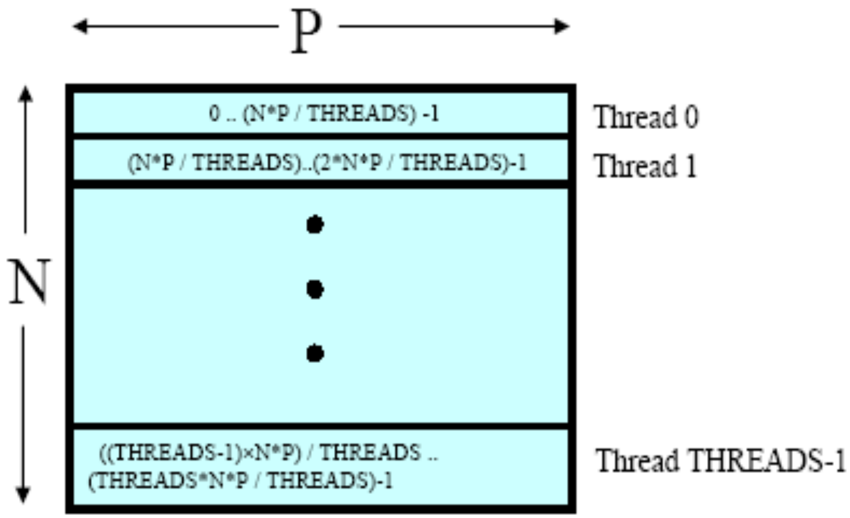
Example con't : sequential C

```
#include <stdlib.h>
#include <time.h>
#define N 4
#define P 4
#define M 4
int a[N][P] = {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N]
   [M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};
void main () {
  int i, j , l;
  for (i = 0 ; i < N; i++) {
    for (j = 0; j < M; j++) {
      c[i][j] = 0;
      for (l = 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b[l][j];
    }
  }
  return 0;
}
```

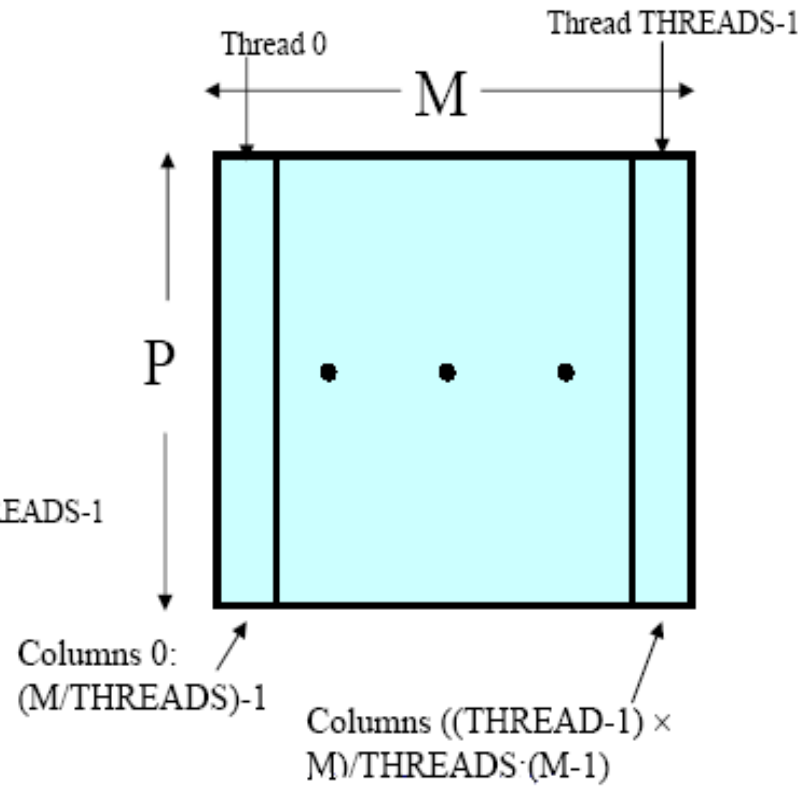
Example: data decomposition in UPC

- Exploits locality in matrix multiplication

- A (N × P) is decomposed row-wise into blocks of size (N × P)/THREADS as shown below:



- B(P × M) is decomposed column wise into M/THREADS blocks as shown below:



•**Note:** N and M are assumed to be multiples of THREADS

Example: UPC code

```
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4
// a, b, and c are blocked shared matrices
// fill in the missing block sizes
shared [                ] int a[N][P] =
{1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}
shared [                ] c[N][M];
shared [                ] int b[P][M] =
   {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};
int main () {
  int i, j , l; // private variables
  upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
     for (j=0 ; j<M ;j++) {
       c[i][j] = 0;
       for (l= 0 ; l<P ; l++)
          c[i][j] += a[i][l]*b[l][j];
    }
  }
  return 0;
}
```

Example: UPC code w/block copy

```
#include <upc_relaxed.h>
/* Assume same shared variables as before */

int b_local[P][M]; //local global variable

int main () {
  int i, j , l; // private variables
  upc_memget(b_local, b, P*M*sizeof(int));

  upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
    for (j=0 ; j<M ;j++) {
      c[i][j] = 0;
      for (l= 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b_local[l][j]; // now local
      }
  }
  return 0;
}
```

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

UPC Libraries

- **UPC Collective Library**
- **UPC-IO Library**

Overview of UPC Collectives

- A collective function performs an operation in which all threads participate.
- Recall that UPC includes the collectives
  - upc_barrier, upc_notify, upc_wait,upc_all_alloc, upc_all_lock_alloc
- Collective Library includes functions for parallel bulk data movement and computation.
  - upc_all_broadcast, upc_all_exchange, upc_all_prefix_reduce, etc.
  - Provides ways to send, gather, exchange, permute, sort, reduce, perform arithmetic operations, etc. on shared data

UPC Collective Example – upc_all_reduce

Syntax:

```
void upc_all_reduceT(shared void *dst, shared const void *src, upc_op_t op,
    size_t nelems, size_t blk size, TYPE (*func)(TYPE, TYPE), upc_flag_t
    sync_mode);
```

Example:

```
#define BLK_SIZE 3
#define NELEMS 10
shared [BLK_SIZE] long A[NELEMS*THREADS];
shared long B;
long result;
// Initialize A. The result below is defined only on thread 0.
upc_barrier;
upc_all_reduceL( &B, A, UPC_ADD, NELEMS*THREADS, BLK_SIZE, NULL,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;
result = B;
```

Overview of UPC-IO Library

- Effort by the I/O working group to provide users with a capability to utilize the underlying parallel I/O file system
- Most UPC-IO functions are collective
  - Function entry/exit includes implicit synchronization
  - Single return values for specific functions
- API provided through extension libraries
- UPC-IO data operations support
  - Shared or private buffers
  - Blocking (upc_all_fread_shared(),...)
  - Non-blocking (async) operations (upc_all_fread_shared_async(), ...)
- Supports List-IO Access
- Several reference implementations by GWU
- Not yet part of standard

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

UPC optimizations

- Space privatization: use pointer-to-locals instead of pointer-to-shareds when dealing with local shared data (through casting and assignments)
- Block moves: use block copy instead of copying elements one by one with a loop, through string operations or structures
- Latency hiding: overlap remote accesses with local processing using split-phase barriers
- Finally, data layout can be key to overall program performance (strive to minimize remote data accesses by keeping data close to computation)

UPC optimizations: local pointers to shared

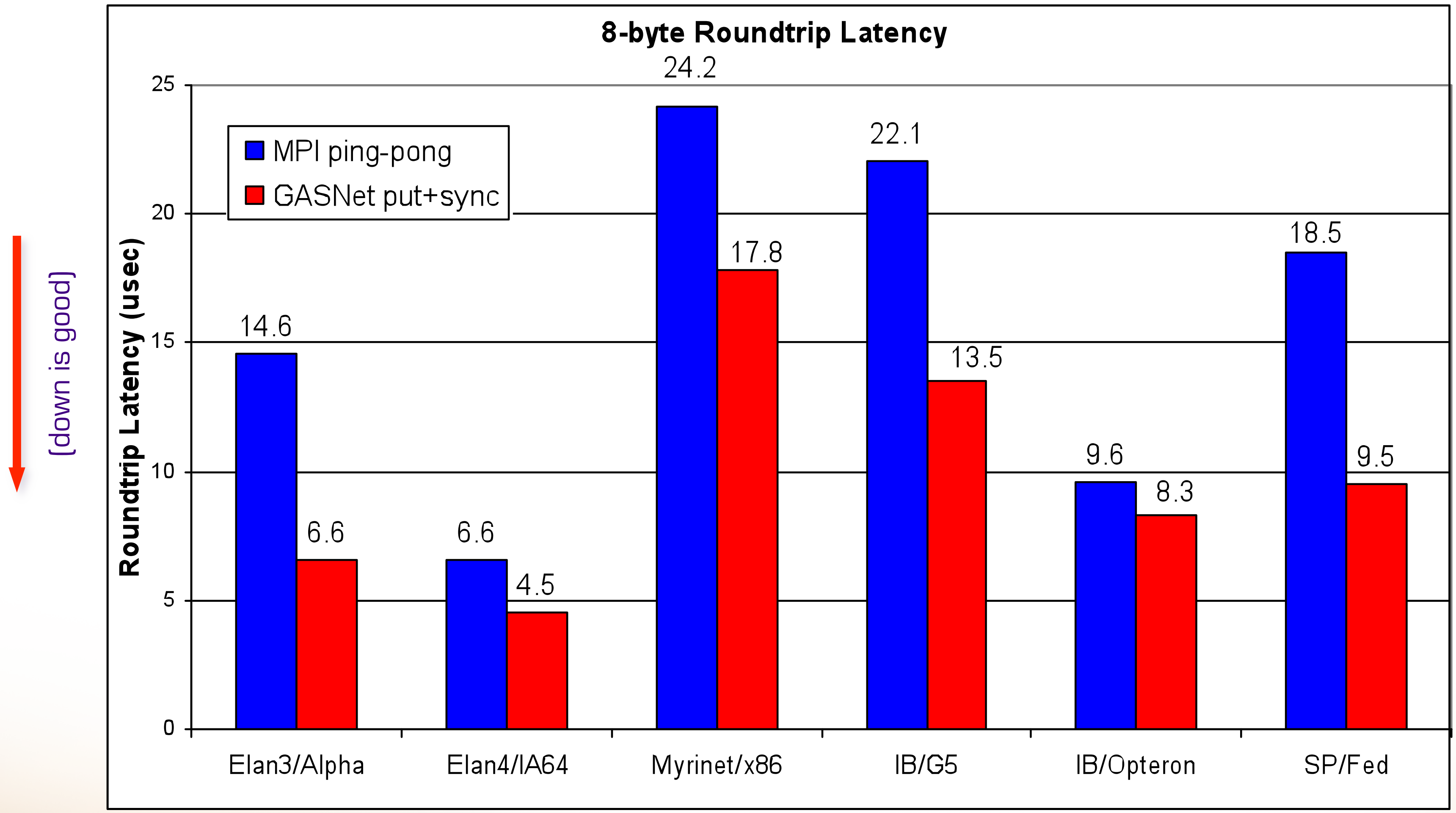
```
…
int *pa = (int*) &A[i][0]; //A and C are declared as shared
int *pc = (int*) &C[i][0];
…
upc_forall(i=0;i<N;i++;&A[i][0])
{
 for(j=0;j<P;j++)
      pa[j]+=pc[j];
}
```

- Pointer arithmetic is faster using local pointers than pointer to shared.
- The pointer dereference can be one order of magnitude faster.

Keys to PGAS Performance

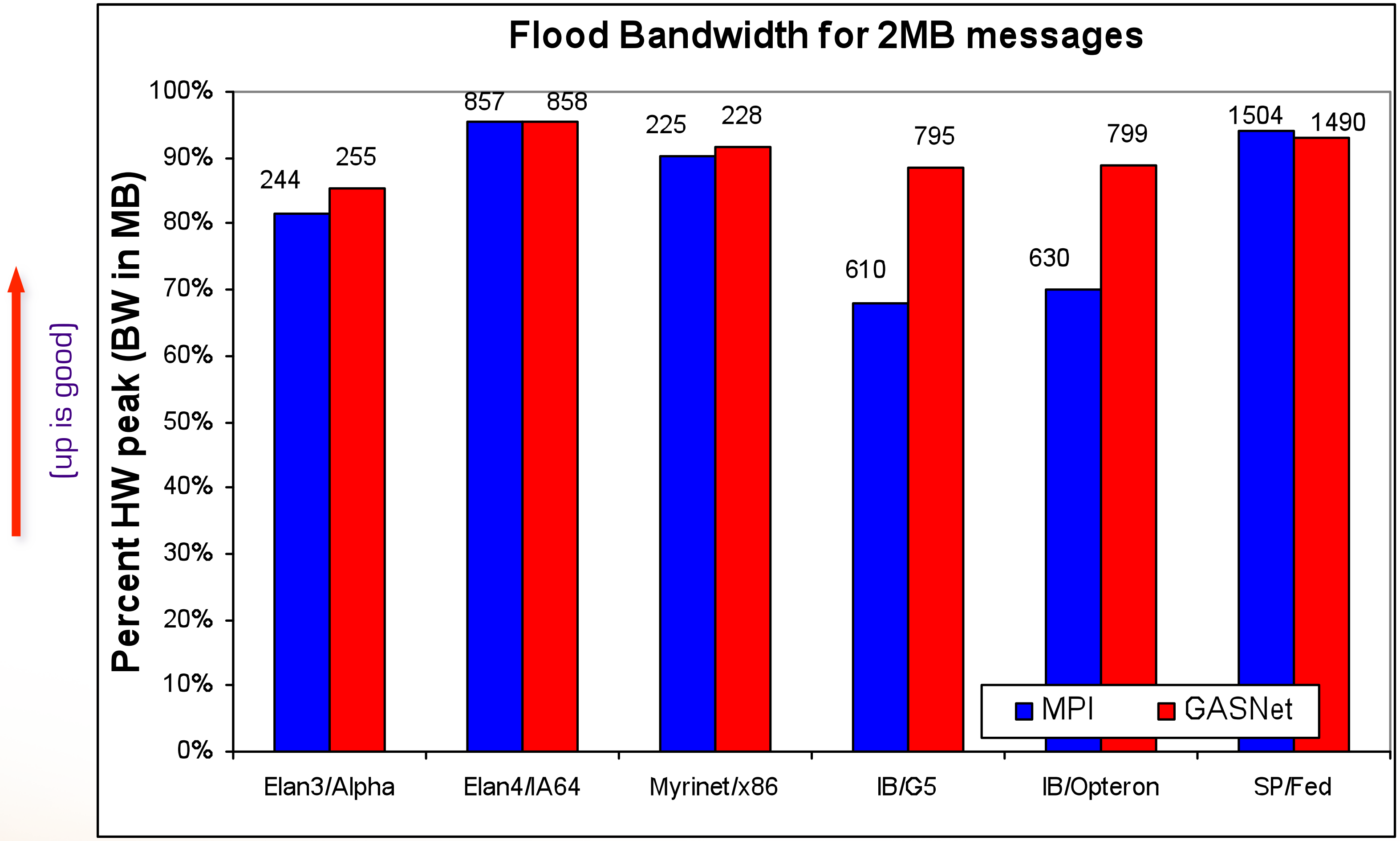
- Parallelism
  - Scaling the number of processors
- Maximize single node performance
  - Generate friendly code or use tuned libraries (BLAS, FFTW, etc.)
- Avoid (unnecessary) communication cost
  - Latency, bandwidth, overhead
  - Berkeley UPC and Titanium use GASNet communication layer
- Avoid unnecessary delays due to dependencies
  - Load balance; Pipeline algorithmic dependencies
- Parallel Performance Wizard (PPW)
  - Performance analysis tool for PGAS programs
  - http://ppw.hcs.ufl.edu/

GASNet: Portability *and* High-Performance



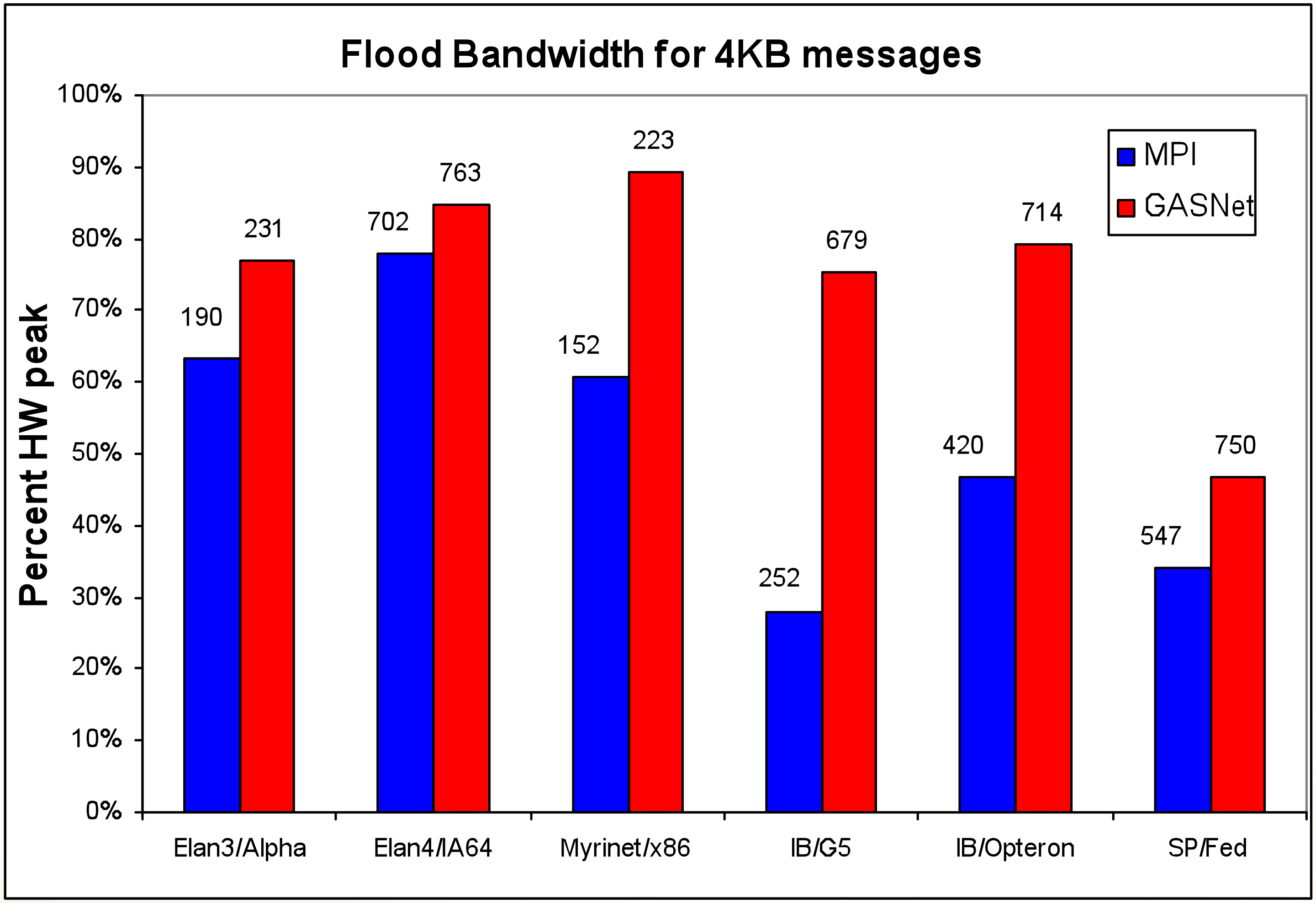
GASNet better for latency across machines

GASNet: Portability *and* High-Performance (2)



GASNet at least as high (comparable) for large messages

GASNet: Portability *and* High-Performance (3)

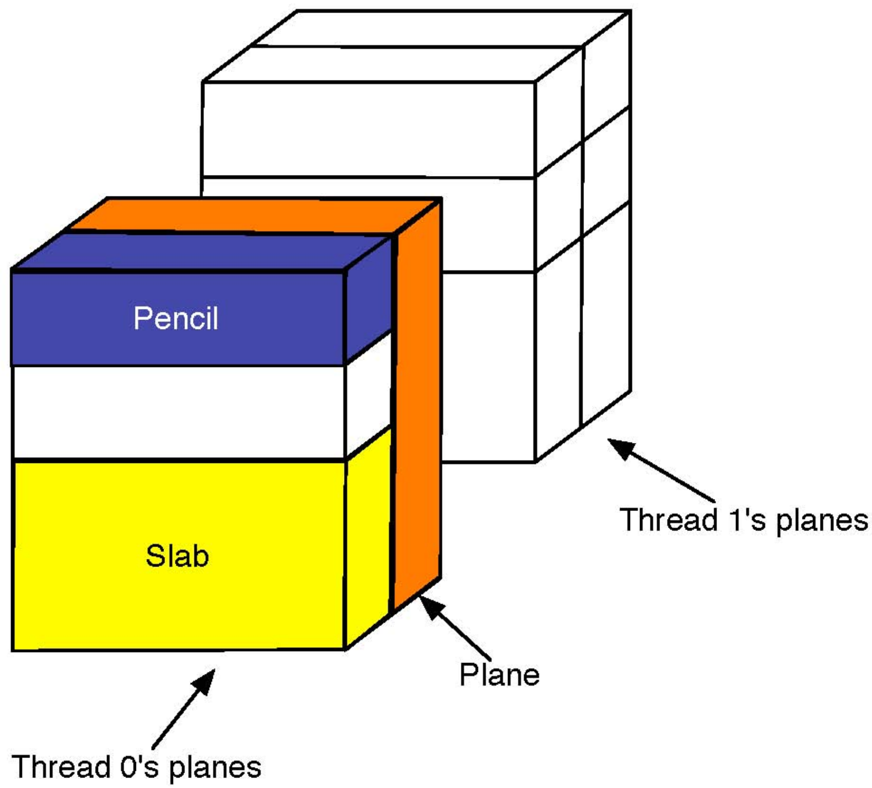


[up is good]

GASNet excels at mid-range sizes: important for overlap

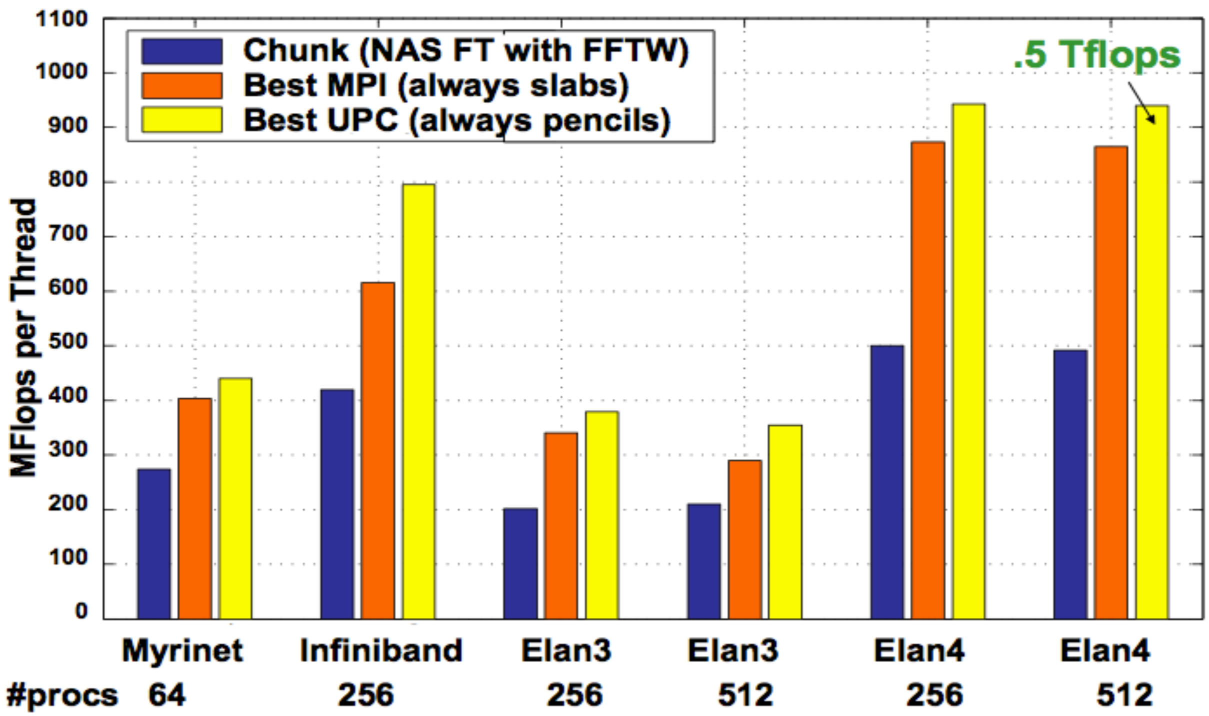
Case Study: NAS FT

- Performance of Exchange (Alltoall) is critical
  - 1D FFTs in each dimension, 3 phases
  - Transpose after first 2 for locality
  - Bisection bandwidth-limited
    - Problem as #procs grows
- Three approaches:
  - Exchange:
    - wait for 2^{nd} dim FFTs to finish, send 1 message per processor pair
  - Slab:
    - wait for chunk of rows destined for 1 proc, send when ready
  - Pencil:
    - send each row as it completes

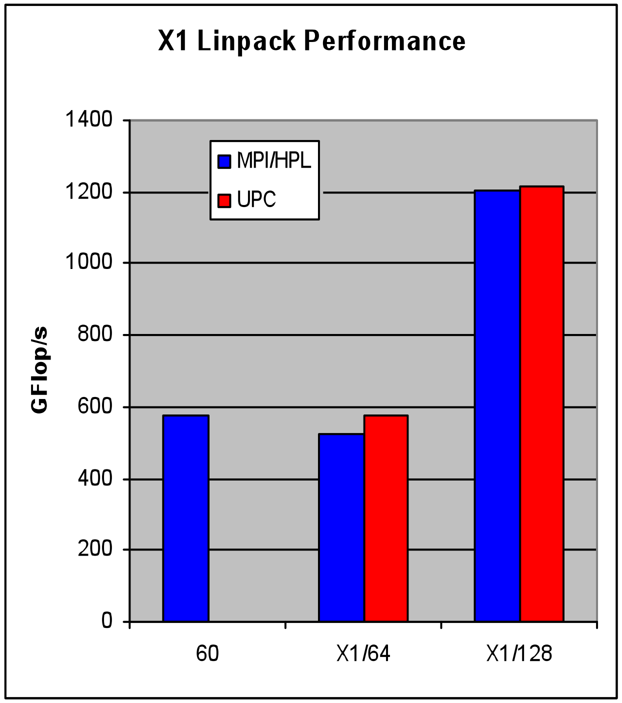


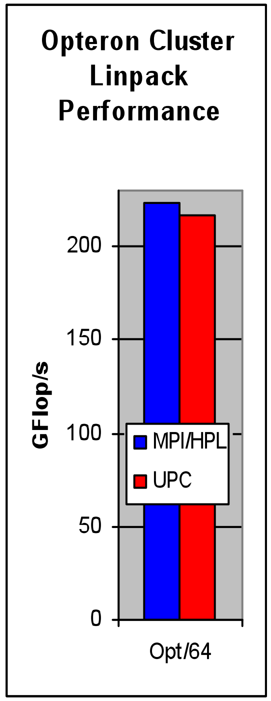
FFT Performance Comparison

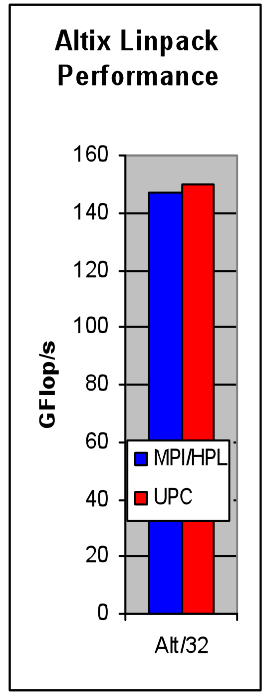
Comparison of 3D FFT performance across several machines using a bulk-synchronous MPI implementation that minimizes message counts but precludes overlap, an MPI code that uses overlap, and a UPC code that uses finer-grained overlap and smaller messages (from Yelick et.al., "Productivity and performance using partitioned global address space languages" in *Proc. 2007 Intl. Workshop on Parallel Symbolic Computation (PASCO '07)*



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)

Outline of talk

1. PGAS Background
2. UPC Background
3. UPC memory/execution model
4. Data and pointers
5. Dynamic memory management
6. Work distribution/synchronization
7. Memory consistency model
8. Programming example
9. UPC libraries
10. Performance tuning
11. Summary

Summary

- UPC is easy to program in for C writers, significantly easier than alternative paradigms at times.
- UPC performance compares favorably with MPI.
  - On some systems, performance of UPC can even be much better
  - Latency hiding and bandwidth optimization of compilers still weak.
- Hand tuned code, with block moves, is still substantially simpler than message passing code.
  - Language and runtime system take care of boring/repetitive communication details.

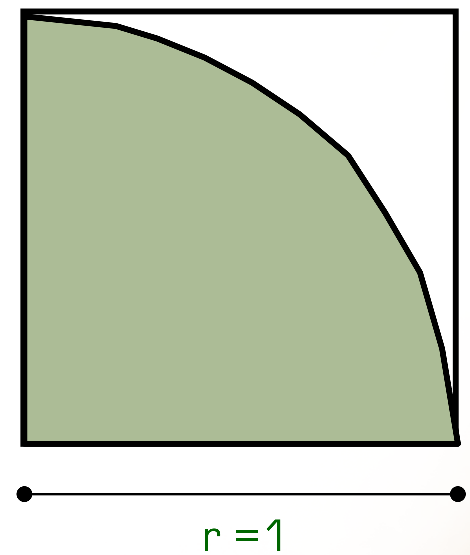
Group Exercise: Matrix-vector multiplication

```
// vect_mat_mult.c
#include <upc_relaxed.h>
shared int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];
void main (void)
{
   int i, j;
   upc_forall( i = 0 ; i < THREADS ; i++; i) {
        c[i] = 0;
        for (j=0; j < THREADS; j++)
            c[i] += a[i][j]*b[j];
        }
}
```

Is the above the best data distribution for this operation?
If not, what would be better and how would you change the code?

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4*\text{ratio}$



Helper Code for Pi in UPC

- Required includes:

```
#include <stdio.h>
#include <math.h>
#include <upc_relaxed.h>
```

- Function to throw dart and calculate where it hits:

```
int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
       return(1);
  } else {
       return(0);
  }
}
```

Pi in UPC: Shared Memory Style

- **Parallel computing of pi, but with a bug**

```
shared int hits;
main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
      hits += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
      printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

shared variable to record hits

divide work up evenly

accumulate hits

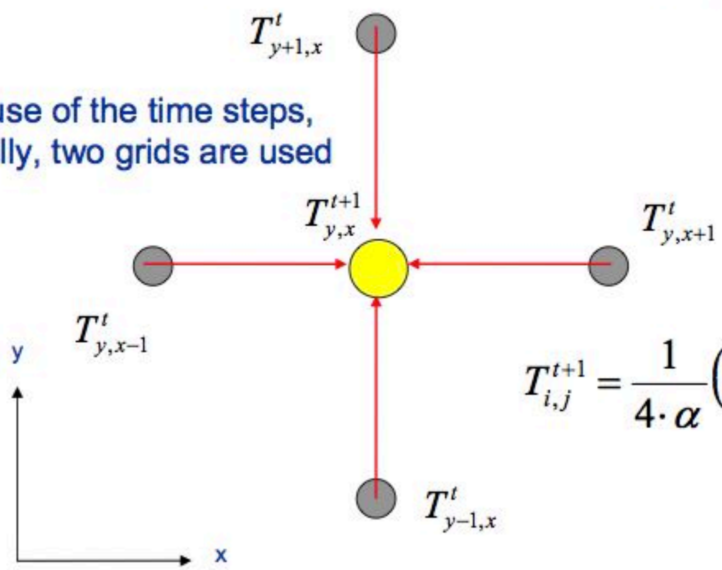
Group exercise: What is the problem with this program and how can we fix it?

Homework problem 3

2D Heat Conduction Problem

- **Based on the 2D Partial Differential Equation (1), 2D Heat Conduction problem is similar to a 4-point stencil operation, as seen in (2):**

Because of the time steps,
Typically, two grids are used



$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha}\frac{\partial T}{\partial t} \quad (1)$$

$$T^{t+1}_{i,j} = \frac{1}{4\cdot\alpha}\left(T^{t}_{i-1,j} + T^{t}_{i+1,j} + T^{t}_{i,j-1} + T^{t}_{i,j+1}\right) \quad (2)$$

Homework problem 3 (cont.)