Parallel Programming Using A Distributed Shared Memory Model

William Carlson - IDA/CCS
Tarek El-Ghazawi - GWU
Robert Numrich - Cray, Inc.
Katherine Yelick - UC Berkeley

Outline of the Day

- Introduction to Distributed Shared Memory
- UPC Programming
- Lunch
- Co-Array Fortran Programming
- Titanium Programming
- Summary

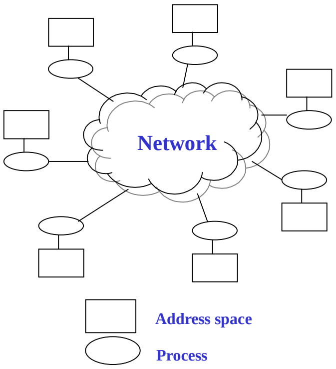
Outline of this Talk

- Basic Concepts
  - Applications
  - Programming Models
  - Computer Systems
- The Program View
- The Memory View
- Synchronization
- Performance AND Ease of Use

Parallel Programming Models

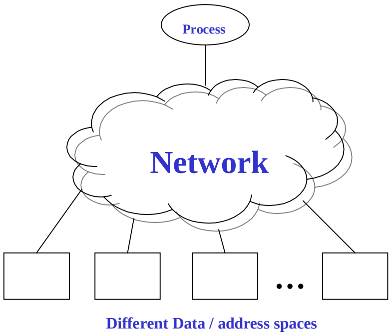
- What is a programming model?
  - A view of data and execution
  - Where architecture and applications meet
- Best when a "contract"
  - Everyone knows the rules
  - Performance considerations important
- Benefits
  - Application - independence from architecture
  - Architecture - independence from applications

The Message Passing Model



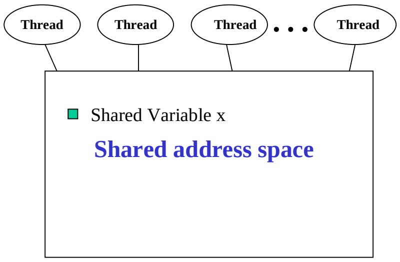
- Programmers control data and work distribution
- Explicit communication
- Significant communication overhead for small transactions
- Example: MPI

The Data Parallel Model



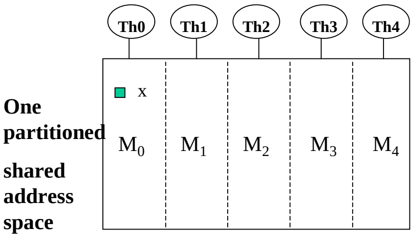
- Easy to write and comprehend, no synchronization required
- No independent branching

The Shared Memory Model



- Simple statements
  - read remote memory via an expression
  - write remote memory through assignment
- Manipulating shared data may require synchronization
- Does not allow locality exploitation
- Example: OpenMP

The Distributed Shared Memory Model

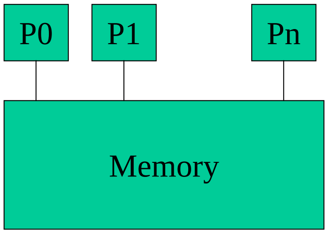


- Similar to the shared memory paradigm
- Memory M_i has affinity to thread Th_i
- Helps exploiting locality of references
- Simple statements
- Examples: This Tutorial!

Tutorial Emphasis

- Concentrate on Distributed Shared Memory Programming as a universal model
  - UPC
  - Co-Array Fortran
  - Titanium
- Not too much on hardware or software support for DSM after this talk...

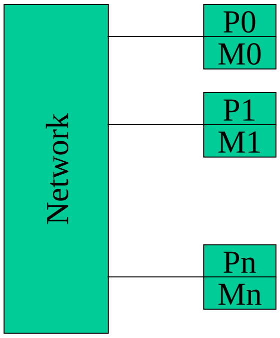
How to share an SMP



- Pretty easy - just map
  - Data to memory
  - Threads of computation to
    - Pthreads
    - Processes
- NUMA vs. UMA
- Single processor is just a virtualized SMP

How to share a DSM

- Hardware models
  - Cray T3D/T3E
  - Quadrics
  - InfiniBand
- Message passing
  - IBM SP (LAPI)



How to share a Cluster

- What is a cluster
  - Multiple Computer/Operating System
  - Network (dedicated)
- Sharing Mechanisms
  - TCP/IP Networks
  - VIA/InfiniBand

Some Simple Application Concepts

- Minimal Sharing
  - Asynchronous work dispatch
- Moderate Sharing
  - Physical systems/ “Halo Exchange”
- Major Sharing
  - The “don’t care, just do it” model
  - May have performance problems on some system

History

- Many data parallel languages
- Spontaneous new idea: “global/shared”
  - Split-C -- Berkeley (Active Messages)
  - AC -- IDA (T3D)
  - F-- -- Cray/SGI
  - PC++ -- Indiana
  - CC++ -- ISI

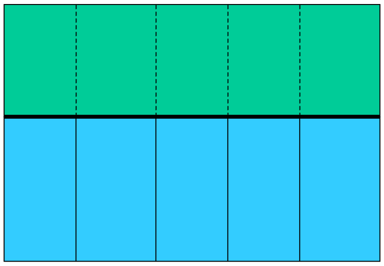
Related Work

- BSP -- Bulk Synchronous Protocol
  - Alternating compute-communicate
- Global Arrays
  - Toolkit approach
  - Includes locality concepts

Model: Program View

- Single “program”
- Multiple threads of control
- Low degree of virtualization
- Identity discovery
- Static vs. Dynamic thread multiplicity

Model: Memory View



- “Shared” area
- “Private” area
- References and pointers
  - – Only “local” thread may reference private
  - – Any thread may reference/point to shared

Model: Memory Pointers and Allocation

- A pointer may be
  - – private
  - – shared
- A pointer may point to:
  - – local
  - – global
- Need to allocate both private and shared
- Bootstrapping

Model: Program Synchronization

- Controls relative execution of threads
- Barrier concepts
  - – Simple: all stop until everyone arrives
  - – Sub-group barriers
- Other synchronization techniques
  - – Loop based work sharing
  - – Parallel control libraries

Model: Memory Consistency

- Necessary to define semantics
  - – When are "accesses" "visible"?
  - – What is relation to other synchronization?
- Ordering
  - – Thread A does two stores
    - Can thread B see second before first?
    - Is this good or bad?

Model: Memory Consistency

- Ordering Constraints
  - – Necessary for memory based synchronization
    - lock variables
    - semaphores
  - – Global vs. Local constraints
- Fences
  - – Explicit ordering points in memory stream

Performance AND Ease of Use

- Why explicit message passing is often bad
- Contributors to performance under DSM
- Some optimizations that are possible
- Some implementation strategies

Why not a Message Passing Model

- Message passing as a mechanism is great
- In some cases it is a good match
  - – DNS (or “the net” application)
- Currently the most portable
- Many applications don’t map so well
  - – Math/Science apps
  - – Data Mining

Contributors to Performance

- Match between architecture and model
  - – If match is poor, performance can suffer greatly
    - Try to send single word messages on Ethernet
    - Try for full memory bandwidth with message passing
- Match between application and model
  - – If model is too strict, hard to express
    - Try to express a linked list in data parallel

Architecture ⇔ Model Issues

- Make model match many architectures
  - Distributed
  - Shared
  - Non-Parallel
- No machine-specific models
- Promote performance potential of all
  - Marketplace will work out value

Application ⇔ Model Issues

- Start with an expressive model
  - Many applications
  - User productivity/debugging
- Performance
  - Don’t make model too abstract
  - Allow annotation

Just a few optimizations possible

- Reference combining
- Compiler/runtime directed caching
- Remote memory operations

Implementation Strategies

- Hardware sharing
  - Map threads onto processors
  - Use existing sharing mechanisms
- Software sharing
  - Map threads to pthreads or processes
  - Use a runtime layer to communicate

Conclusions

- Using distributed shared memory is good
- Questions?
- Enjoy the rest of the tutorial

Programming in UPC

Tarek El-Ghazawi

The George Washington University
tarek@seas.gwu.edu

UPC Outline

1. **Background and Philosophy**
2. **UPC Execution Model**
3. **UPC Memory Model**
4. **UPC: A Quick Intro**
5. **Data and Pointers**
6. **Dynamic Memory Management**
7. **Programming Examples**
8. **Synchronization**
9. **Performance Tuning and Early Results**
10. **Concluding Remarks**

What is UPC?

- Unified Parallel C
- An explicit parallel extension of ANSI C
- A distributed shared memory parallel programming language

Design Philosophy

- Similar to the C language philosophy
  - Programmers are clever and careful
  - Programmers can 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

Road Map

- Start with C, the other proven language besides FORTRAN
- Keep all powerful C concepts and features
- Add parallelism, learn from Split-C, AC, PCP, etc.
- Integrate user community experience and experimental performance observations
- Integrate developer’s expertise from vendors, government, and academia

⇒ UPC !

History

- Initial Tech. Report from IDA in collaboration with LLNL and UCB in May 1999.
- UPC consortium of government, academia, and HPC vendors coordinated by GWU, IDA, DoD
- The participants currently are: ARSC, Compaq, CSC, Cray Inc., Etnus, GMU, HP, IBM, IDA CSC, Intrepid Technologies, LBNL, LLNL, MTU, NSA, SGI, Sun Microsystems, UCB, US DoD, US DoE

Status

- Specification v1.0 completed February of 2001
- Benchmark, UPC_Bench, v1.0pre1
- Testing suite v1.0
- 2-Day Course offered in the US and abroad
- Research Exhibits at SC 2000 and SC 2001[R547]
- UPC web site: upc.gwu.edu
- UPC Book by SC 2002?

Hardware Platforms

- UPC implementations are available for
  - Cray T3D/E
  - Compaq AlphaServer SC
  - SGI O 2000
- Ongoing and future implementations for:
  - HP
  - Sun multiprocessors
  - Cray SV-2
  - IBM
  - Beowulf Clusters

UPC Outline

1. **Background and Philosophy**
2. **UPC Execution Model**
3. **UPC Memory Model**
4. **UPC: A Quick Intro**
5. **Data and Pointers**
6. **Dynamic Memory Management**
7. **Programming Examples**
8. **Synchronization**
9. **Performance Tuning and Early Results**
10. **Concluding Remarks**

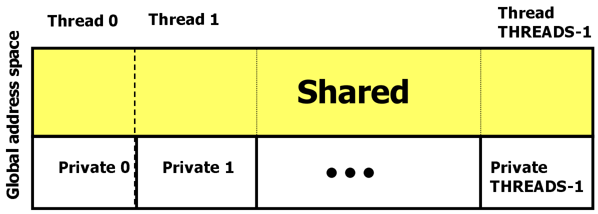
UPC Execution Model

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

UPC Outline

1. **Background and Philosophy**
2. **UPC Execution Model**
3. **UPC Memory Model**
4. **UPC: A Quick Intro**
5. **Data and Pointers**
6. **Dynamic Memory Management**
7. **Programming Examples**
8. **Synchronization**
9. **Performance Tuning and Early Results**
10. **Concluding Remarks**

UPC Memory Model



- A shared pointer can reference all locations in the shared space
- A private pointer may reference only 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

---

User’s General View

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

UPC Outline

1. Background and Philosophy
2. UPC Execution Model
3. UPC Memory Model
4. UPC: A Quick Intro
5. Data and Pointers
6. Dynamic Memory Management
7. Programming Examples
8. Synchronization
9. Performance Tuning and Early Results
10. Concluding Remarks

A First Example: Vector addition

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];
void main(){
    int i;
    for(i=0; i<N; i++)
            If (MYTHREAD==i%THREADS)
                        v1plusv2[i]=v1[i]+v2[i];
}
```

2nd Example: Vector Addition with upc_forall

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS

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];
}
```

Compiling and Running on Cray

- **Cray**
  - To compile with a fixed number (4) of threads:
    - upc –O2 –fthreads-4 –o vect_add vect_add.c
  - To run:
    - ./vect_add

Compiling and Running on Compaq

- **Compaq**
  - To compile with a fixed number of threads and run:
    - upc –O2 –fthreads 4 –o vect_add vect_add.c
    - prun ./vect_add
  - To compile without specifying a number of threads and run:
    - upc –O2 –o vect_add vect_add.c
    - prun –n 4 ./vect_add

UPC DATA:
Shared Scalar and Array Data

- The shared qualifier, a new qualifier
- 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 on system (affinity to thread 0) */
int b; /* one private b at each thread */
```

UPC Pointers

- Pointer declaration:

  `shared int *p;`

- p is a pointer to
  an integer residing in the shared memory space.
- p is called a shared pointer.

Shared Pointers:A Third Example

```
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
        int i;
        shared int *p1, *p2;

        p1=v1; p2=v2;
        upc_forall(i=0; i<N; i++, p1++, p2++; i)
                v1plusv2[i]=*p1+*p2;
}
```

Synchronization - Barriers

- No implicit synchronization among the threads
- Among the synchronization mechanisms offered by UPC are:
  - Barriers (Blocking)
  - Split Phase Barriers
  - Locks

Work Sharing with upc_forall()

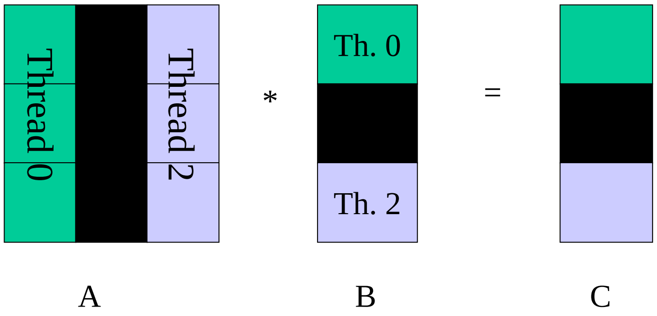
- Distributes independent iterations
- Each thread gets a bunch of iterations
- Affinity (expression) field to distribute work
- Simple C-like syntax and semantics

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

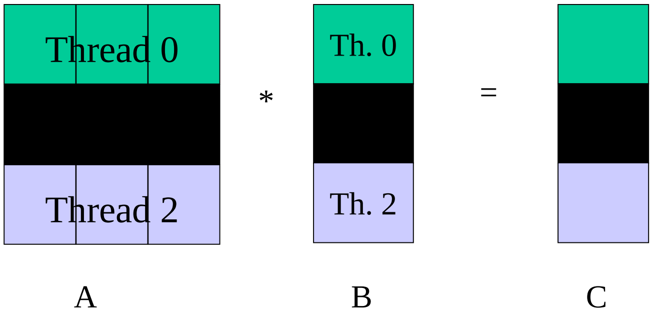
Example 4: UPC Matrix-Vector Multiplication- Default Distribution

```
// vect_mat_mult.c

#include <upc_relaxed.h>

shared int a[THREADS][THREADS], c[THREADS];
shared int b[THREADS];

void main (void) {
        int i, j , l;

        upc_forall( i = 0 ; i < THREADS ; i++; i) {
                c[i] = 0;
                for ( l= 0 ; l< THREADS ; l++)
                        c[i] += a[i][l]*b[l];
        }
}
```

Data Distribution



A Better Data Distribution



Example 5: UPC Matrix-Vector Multiplication-- The Better Distribution

```
// vect_mat_mult.c

#include <upc_relaxed.h>

shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void) {
        int i, j , l;

        upc_forall( i = 0 ; i < THREADS ; i++; i) {
                c[i] = 0;
                for ( l= 0 ; l< THREADS ; l++)
                        c[i] += a[i][l]*b[l];
        }
}
```

UPC Outline

1. Background and Philosophy
2. UPC Execution Model
3. UPC Memory Model
4. UPC: A Quick Intro
5. Data, Pointers, and Work Sharing
6. Dynamic Memory Management
7. Programming Examples
8. Synchronization
9. Performance Tuning and Early Results
10. Concluding Remarks

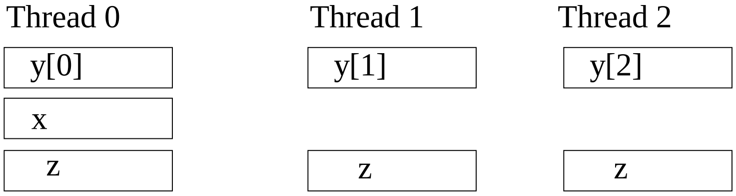
Shared and Private Data

Examples of Shared and Private Data Layout:

Assume THREADS = 3

```
shared int x;  /*x will be aligned with 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:

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]

Shared and Private Data

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

will result in the following data layout:

Thread 0	Thread 1	•••	Thread (THREADS-1)
A[0][0]	A[0][1]	•••	A[0][THREADS-1]
A[0][THREADS]	A[0][THREADS+1]		A[0][2*THREADS-1]
A[1][0]	A[1][1]		A[1][THREADS-1]
A[1][THREADS]	A[1][THREADS+1]		A[1][2*THREADS-1]

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

- 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:

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

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

Shared and Private Data

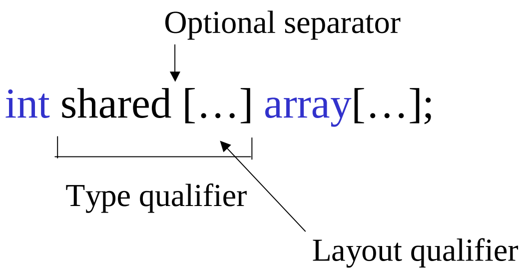
Assume THREADS = 4

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

will result in the following data layout:

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]			

Spaces and Parsing of the Shared Type Qualifier: **As Always in C Spacing Does Not Matter!**



Optional separator

int shared […] array[…];

Type qualifier

Layout qualifier

UPC Pointers

Where does the pointer reside?

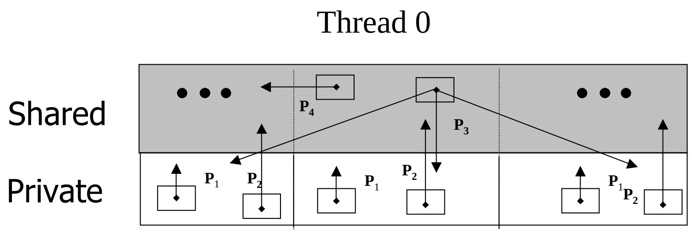
		Private	Shared
Where does it point?	Private	PP	PS
	Shared	SP	SS

UPC Pointers

- How to declare them?
  - int *p1; /* private pointer pointing locally */
  - shared int *p2; /* private pointer pointing into the shared space */
  - int *shared p3; /* shared pointer pointing locally */
  - shared int *shared p4; /* shared pointer pointing into the shared space */

- As a convention, “shared pointer” means a pointer pointing to a shared object. It generally means an equivalent of p2 but could be p4.

UPC Pointers



UPC Pointers

- What are the common usages?
  - int *p1; /* access to private data */
  - shared int *p2; /* access of local thread to data in shared space */
  - int *shared p3; /* not recommended*/
  - shared int *shared p4; /* access of all threads to data in the shared space*/

UPC Pointers

- In UPC for Cray T3E , pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)
- Example: Cray T3E implementation

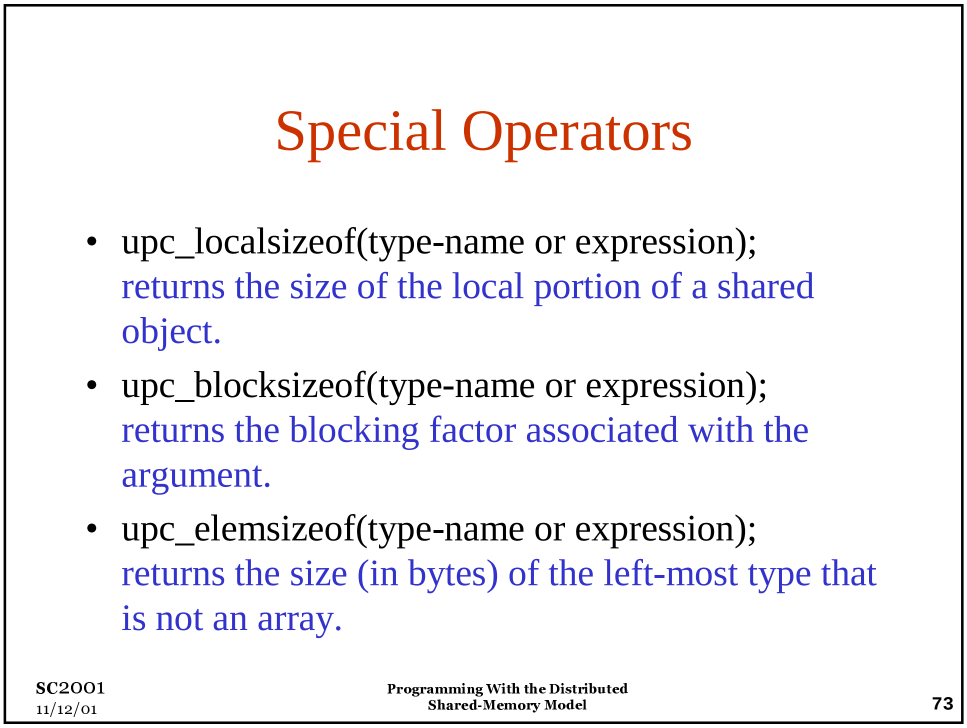
Phase	Thread	Virtual Address
63 49	48 38	37 0

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 shared pointer to a private pointer, the thread number of the shared pointer may be lost
- Casting of shared to private is well defined only if the shared pointer has affinity with the thread performing the cast

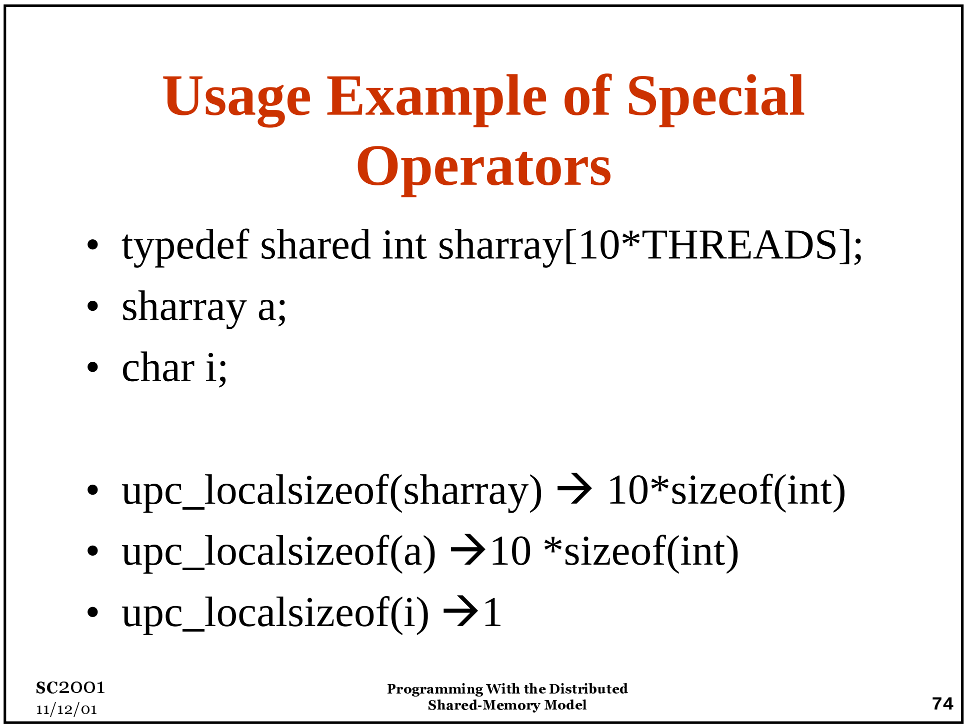
Special Functions

- int upc_threadof(shared void *ptr);
  returns the thread number that has affinity to the shared pointer
- int upc_phaseof(shared void *ptr);
  returns the index (position within the block)field of the shared pointer
- void* upc_addrfield(shared void *ptr);
  returns the address of the block which is pointed at by the shared pointer



Special Operators

- upc_localsizeof(type-name or expression);
  returns the size of the local portion of a shared object.
- upc_blocksizeof(type-name or expression);
  returns the blocking factor associated with the argument.
- upc_elemsizeof(type-name or expression);
  returns the size (in bytes) of the left-most type that is not an array.



Usage Example of Special Operators

- typedef shared int sharray[10*THREADS];
- sharray a;
- char i;

- upc_localsizeof(sharray) → 10*sizeof(int)
- upc_localsizeof(a) →10 *sizeof(int)
- upc_localsizeof(i) →1

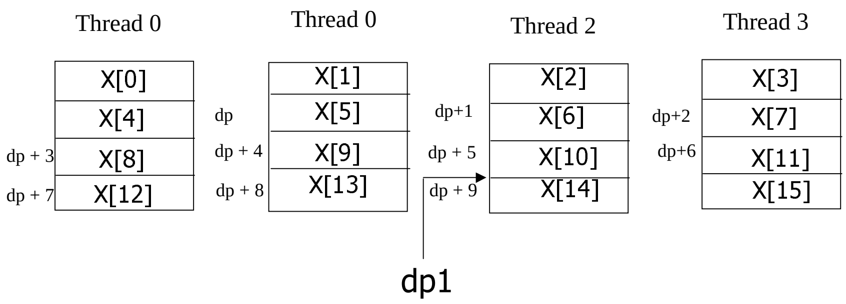
UPC Pointers

Shared Pointer Arithmetic Examples:

Assume THREADS = 4

```
#define N 16
shared int x[N];
shared int *dp=&x[5], *dp1;
dp1 =  dp + 9;
```

UPC Pointers



	Thread 0		Thread 0		Thread 2		Thread 3
	X[0]		X[1]		X[2]		X[3]
	X[4]	dp	X[5]	dp+1	X[6]	dp+2	X[7]
dp + 3	X[8]	dp + 4	X[9]	dp + 5	X[10]	dp+6	X[11]
dp + 7	X[12]	dp + 8	X[13]	dp + 9	X[14]		X[15]

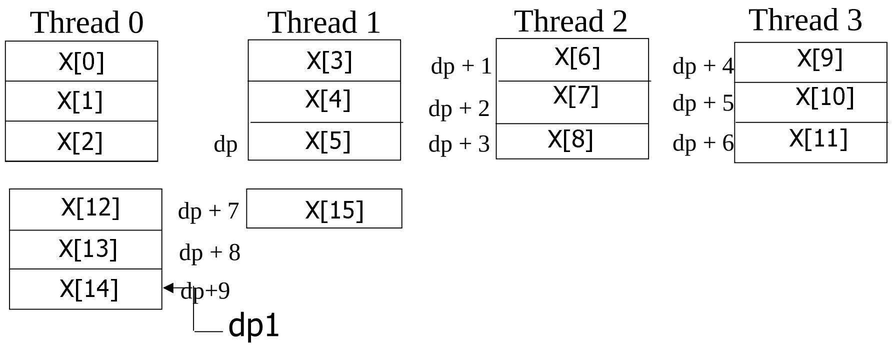
dp1

UPC Pointers

Assume THREADS = 4

```
shared[3] x[N], *dp=&x[5], *dp1;
dp1 =  dp + 9;
```

UPC Pointers



Thread 0		Thread 1		Thread 2		Thread 3
X[0]		X[3]	dp + 1	X[6]	dp + 4	X[9]
X[1]		X[4]	dp + 2	X[7]	dp + 5	X[10]
X[2]	dp	X[5]	dp + 3	X[8]	dp + 6	X[11]
X[12]	dp + 7	X[15]				
X[13]	dp + 8					
X[14]	dp+9					

dp1

UPC Pointers

Example Pointer Castings and Mismatched Assignments:

```
shared  int x[THREADS];
int *p;
p = (int *) &x[MYTHREAD];   /* p points  to x[MYTHREAD] */
```

- Each of the private pointers will point at the x element which has affinity with its thread, i.e. MYTHREAD

UPC Pointers

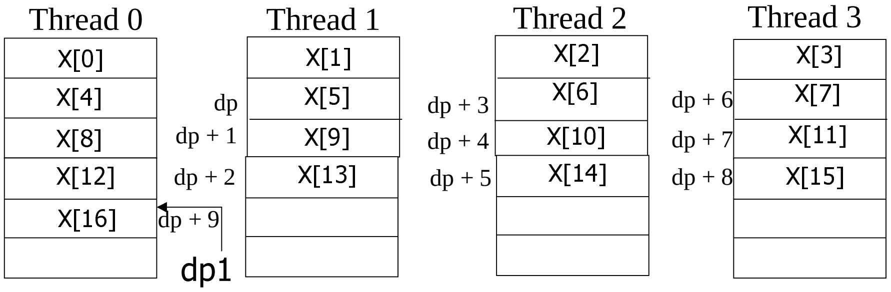
Assume THREADS = 4

```
shared int x[N];
shared[3] int *dp=&x[5], *dp1;
dp1 =  dp + 9;
```

•This statement assigns to dp1 a value that is 9 positions beyond dp

•The pointer will follow its own blocking and not the one of the array

UPC Pointers



Thread 0		Thread 1		Thread 2		Thread 3
X[0]		X[1]		X[2]		X[3]
X[4]	dp	X[5]	dp + 3	X[6]	dp + 6	X[7]
X[8]	dp + 1	X[9]	dp + 4	X[10]	dp + 7	X[11]
X[12]	dp + 2	X[13]	dp + 5	X[14]	dp + 8	X[15]
X[16]	dp + 9					

UPC Pointers

- Given the declarations

  shared[3] int *p;

  shared[5] int *q;
- Then

  p=q; /* is acceptable (implementation may require explicit cast) */
- Pointer p, however, will obey pointer arithmetic for blocks of 3, not 5 !!
- A pointer cast sets the phase to 0

String functions in UPC

- UPC provides standard library functions to move data to/from shared memory
- Can be used to move chunks in the shared space or between shared and private spaces

---

String 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

Worksharing with upc_forall

- Distributes independent iteration across threads in the way you wish– typically to boost locality exploitation
- Simple C-like syntax and semantics

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

- Expression could be an integer expression or a reference to (address of) a shared object

---

Work Sharing: upc_forall()

- Example 1: Exploiting locality

```
shared int a[100],b[100], c[101];
int i;
upc_forall (i=0; i<100; i++; &a[i])
      a[i] = b[i] * c[i+1];
```

- Example 2: distribution in a round-robin fashion

```
shared int a[100],b[100], c[101];
int i;
upc_forall (i=0; i<100; i++; i)
      a[i] = b[i] * c[i+1];
```

Note: Examples 1 and 2 happened to result in the same distribution

- Example 3: distribution by chunks

```
shared int a[100],b[100], c[101];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
      a[i] = b[i] * c[i+1];
```

i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3

UPC Outline

1. Background and Philosophy
2. UPC Execution Model
3. UPC Memory Model
4. UPC: A Quick Intro
5. Data, Pointers, and Work Sharing
6. Dynamic Memory Management
7. Programming Examples
8. Synchronization
9. Performance Tuning and Early Results
10. Concluding Remarks

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

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]**
- (Not yet implemented on Cray)

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]

Local Memory Allocation

shared void *upc_local_alloc(size_t nblocks, size_t nbytes);

nblocks : number of blocks
nbytes : block size

- Returns a shared memory space with affinity to the calling thread
- Equivalent to :
  shared [ ] char[nblocks * nbytes]

Memory Freeing

void upc_free(shared void *ptr);

- The upc_free function frees the dynamically allocated shared memory pointed to by ptr
- (Not yet implemented on Cray)

UPC Outline

1. **Background and Philosophy**
2. **UPC Execution Model**
3. **UPC Memory Model**
4. **UPC: A Quick Intro**
5. **Data, Pointers, and Work Sharing**
6. **Dynamic Memory Management**
7. **Programming Examples**
8. **Synchronization**
9. **Performance Tuning and Early Results**
10. **Concluding Remarks**

Example: Matrix Multiplication in UPC

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

$$C_{ij} = \sum_{l=1}^{p} A_{il} \times B_{lj}$$

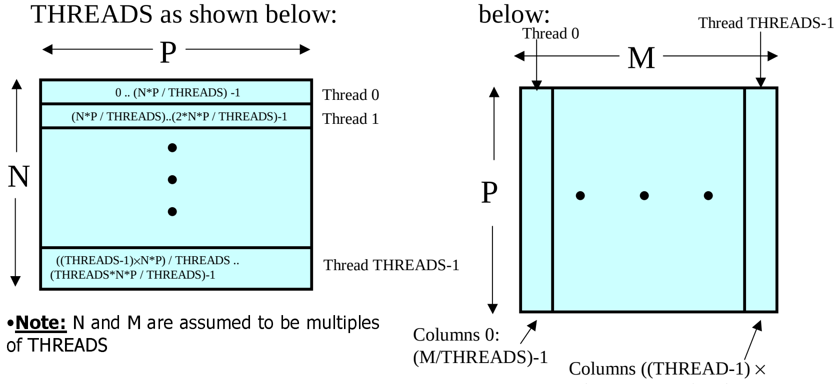
Doing it in C

```
#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 (void) {
    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];
        }
    }
}
```

Domain Decomposition for 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

UPC Matrix Multiplication Code

```
// mat_mult_1.c

#include <upc_relaxed.h>

#define N  4
#define P  4
#define M 4

shared [N*P /THREADS] int a[N][P] =  {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared[M/THREADS] int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

void main (void) {
                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];
                                }
                }
}
```

UPC Matrix Multiplication Code with block copy

```
// mat_mult_3.c

#include <upc_relaxed.h>

shared [N*P /THREADS] int a[N][P], c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared[M/THREADS] int b[P][M];

int b_local[P][M];

void main (void) {
        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];
                }
        }
}
```

Matrix Multiplication with dynamic memory

```
// mat_mult_2.c

#include <upc_relaxed.h>

shared [N*P /THREADS] int *a, *c;
shared[M/THREADS] int *b;

void main (void) {
        int i, j , l; // private variables

        a=upc_all_alloc(N,P*upc_elemsizeof(*a));
        c=upc_all_alloc(N,P* upc_elemsizeof(*c));
        b=upc_all_alloc(M, upc_elemsizeof(*b));

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

Example: Sobel Edge Detection

Original Image

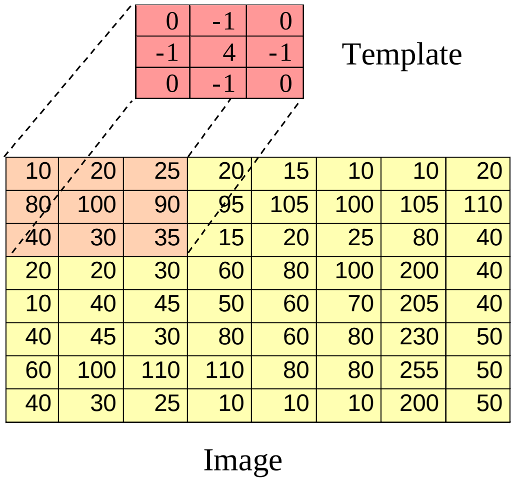
Edge-detected Image

Sobel Edge Detection

- Template Convolution
- Sobel Edge Detection Masks
- Applying the masks to an image

Template Convolution

- The template and the image will do a pixel by pixel multiplication and add up to a result pixel value.
- The generated pixel value will be applied to the central pixel in the resulting image.
- The template will go through the entire image.



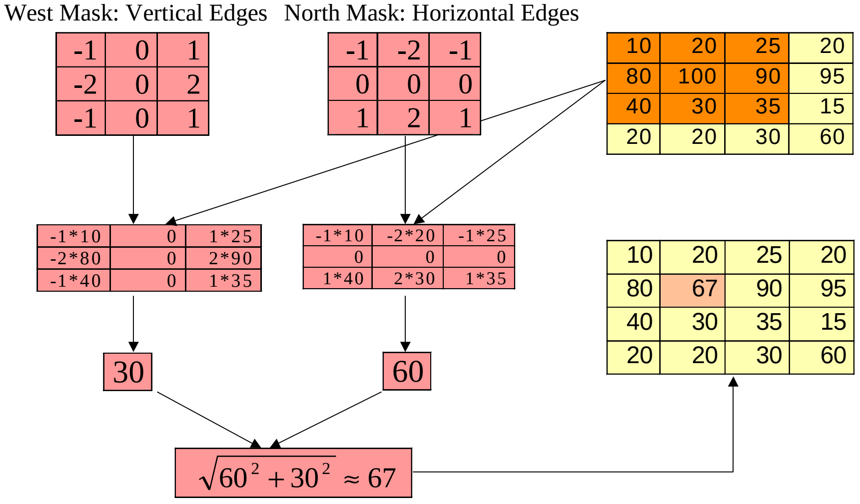
Template

0	-1	0
-1	4	-1
0	-1	0

10	20	25	20	15	10	10	20
80	100	90	95	105	100	105	110
40	30	35	15	20	25	80	40
20	20	30	60	80	100	200	40
10	40	45	50	60	70	205	40
40	45	30	80	60	80	230	50
60	100	110	110	80	80	255	50
40	30	25	10	10	10	200	50

Image

Applying the Masks to an Image



West Mask: Vertical Edges

-1	0	1
-2	0	2
-1	0	1

North Mask: Horizontal Edges

-1	-2	-1
0	0	0
1	2	1

10	20	25	20
80	100	90	95
40	30	35	15
20	20	30	60

-1*10	0	1*25
-2*80	0	2*90
-1*40	0	1*35

-1*10	-2*20	-1*25
0	0	0
1*40	2*30	1*35

10	20	25	20
80	67	90	95
40	30	35	15
20	20	30	60

30

60

$$\sqrt{60^2 + 30^2} \approx 67$$

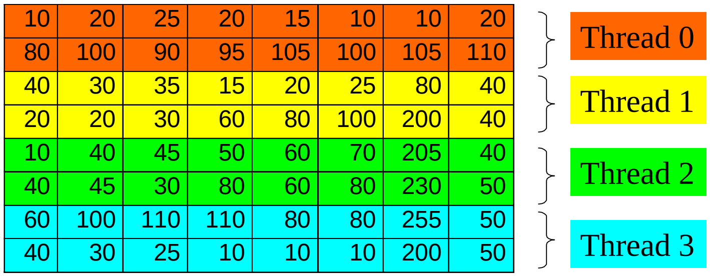
Sobel Edge Detection – The C program

```
#define BYTE unsigned char
BYTE orig[N][N],edge[N][N];
int Sobel()
{  int    i,j,d1,d2;
   double magnitude;
   for (i=1; i<N-1; i++)
   {     for (j=1; j<N-1; j++)
         {       d1  = (int) orig[i-1][j+1] - orig[i-1][j-1];
                 d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;
                 d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];
                 d2  = (int) orig[i-1][j-1] - orig[i+1][j-1];
                 d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;
                 d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];
                 magnitude = sqrt(d1*d1+d2*d2);
                 edge[i][j] = magnitude > 255 ? 255 : (BYTE) magnitude;
         }
   }
   return 0;
}
```

Sobel Edge Detection in UPC

- Distribute data among threads
- Using upc_forall to do the work in parallel

Distribute data among threads



10	20	25	20	15	10	10	20	Thread 0
80	100	90	95	105	100	105	110	
40	30	35	15	20	25	80	40	Thread 1
20	20	30	60	80	100	200	40	
10	40	45	50	60	70	205	40	Thread 2
40	45	30	80	60	80	230	50	
60	100	110	110	80	80	255	50	Thread 3
40	30	25	10	10	10	200	50	

```
shared [16] BYTE orig[8][8],edge[8][8]
```

Or in General

```
shared [N*N/THREADS] BYTE orig[N][N],edge[N][N]
```

Sobel Edge Detection–
The UPC program

```
#define BYTE unsigned char
shared [N*N/THREADS] BYTE orig[N][N],edge[N][N];
int Sobel()
{  int   i,j,d1,d2;
   double magnitude;
   upc_forall (i=1; i<N-1; i++; &edge[i][0])
   {    for (j=1; j<N-1; j++)
        {       d1  = (int) orig[i-1][j+1] - orig[i-1][j-1];
                d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;
                d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];
                d2  = (int) orig[i-1][j-1] - orig[i+1][j-1];
                d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;
                d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];
                magnitude = sqrt(d1*d1+d2*d2);
                edge[i][j] = magnitude > 255 ? 255 : (BYTE) magnitude;
        }
   }
   return 0;
}
```

Notes on the Sobel Example

- Only a few minor changes in sequential C code to make it work in UPC
- N is assumed to be a multiple of THREADS
- Only the first row and the last row of pixels generated in each thread need remote memory reading

UPC Outline

Synchronization

- No implicit synchronization among the threads
- UPC provides the following synchronization mechanisms:
  - Barriers
  - Locks
  - Memory Consistency Control

Synchronization - Barriers

- No implicit synchronization among the threads
- UPC provides the following barrier synchronization constructs:
  - Barriers (Blocking)
    - upc_barrier expr_{opt};
  - Split-Phase Barriers (Non-blocking)
    - upc_notify expr_{opt};
    - upc_wait expr_{opt};

    Note: upc_notify is not blocking upc_wait is

Synchronization - Locks

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

---

Memory Consistency Models

- 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. (no shared operation can begin before the previously specified one is done)

Memory Consistency Models

- 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*
- Programmers responsible for using correct consistency model

---

Memory Consistency

- Default behavior can be controlled by the programmer:
  - Use strict memory consistency

    #include<upc_strict.h>
  - Use relaxed memory consistency

    #include<upc_relaxed.h>

Memory Consistency

- Default behavior can be altered for a variable definition using:
  - Type qualifiers: *strict* & *relaxed*
- Default behavior can be altered for a statement or a block of statements using
  - #pragma upc strict
  - #pragma upc relaxed

UPC Outline

1. **Background and Philosophy**
2. **UPC Execution Model**
3. **UPC Memory Model**
4. **UPC: A Quick Intro**
5. **Data, Pointers, and Work Sharing**
6. **Dynamic Memory Management**
7. **Programming Examples**
8. **Synchronization**
9. **Performance Tuning and Early Results**
10. **Concluding Remarks**

How to Exploit the Opportunities for Performance Enhancement?

- Compiler optimizations
- Run-time system
- Hand tuning

List of Possible Optimizations for UPC Code

1. Space privatization: use private pointers instead of shared pointers when dealing with local shared data (through casting and assignments)
2. Block moves: use block copy instead of copying elements one by one with a loop, through string operations or structures
3. Latency hiding: For example, overlap remote accesses with local processing using split-phase barriers

Typical Performance of Shared vs. Private Accesses

MB/s	read single elements	write single elements
CC	640.0	400.0
UPC Private	686.0	565.0
UPC local shared	7.0	44.0
UPC remote shared	0.2	0.2

Using Local Pointers Instead of Shared Pointers

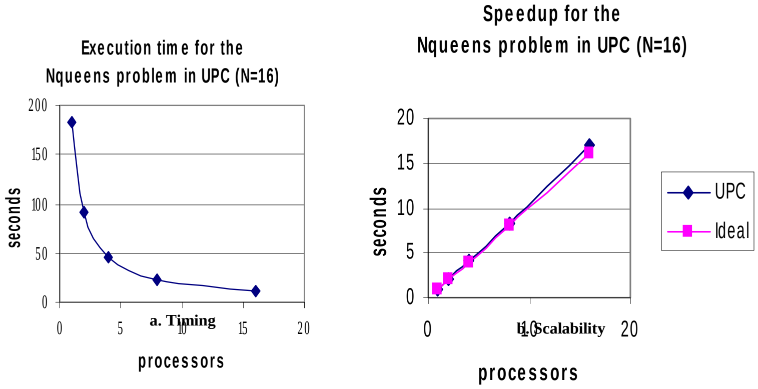
```
…
int *pa = (int*) &A[i][0];
int *pc = (int*) &C[i][0];
…
upc_forall(i=0;i<N;i++;&A[i][0]) {
        for(j=0;j<P;j++)
                pa[j]+=pc[j];
}
```

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

Performance of UPC

- NPB in UPC underway
- Current benchmarking results on Compaq for:
  - Nqueens Problem
  - Matrix Multiplications
  - Sobel Edge detection
  - Synthetic Benchmarks
- Check the web site for a report with extensive measurements on Compaq and T3E

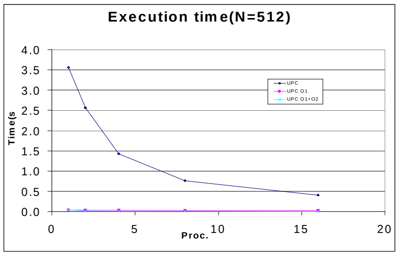
Performance of Nqueens on the Compaq AlphaServer



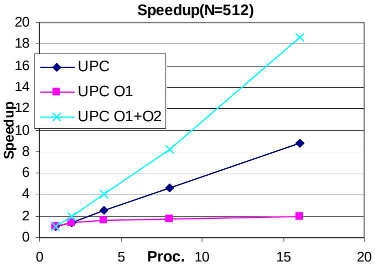
a. Timing

b. Scalability

Performance of Edge detection on the Compaq AlphaServer SC



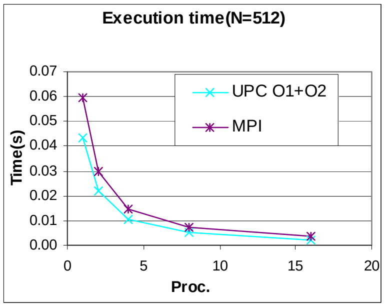
a. Execution time

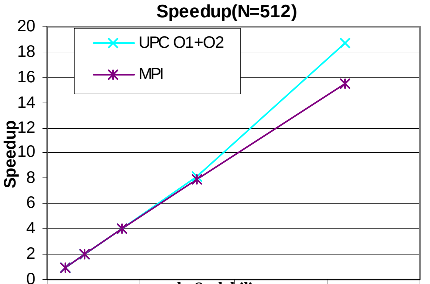


b. Scalability

O1: using private pointers instead of shared pointers
O2: using structure copy instead of element by element

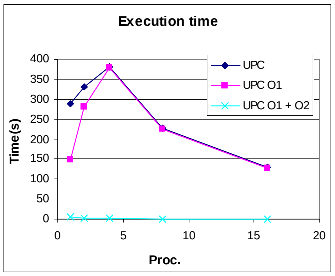
Performance of Optimized UPC versus MPI for Edge detection



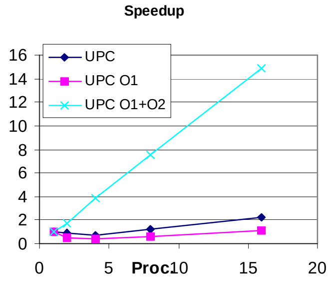


b. Scalability

Effect of Optimizations on Matrix Multiplication on the AlphaServer SC



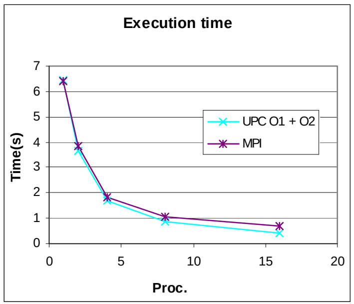
a. Execution time



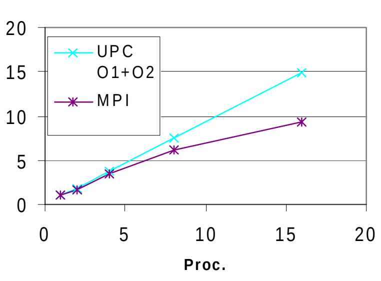
b. Scalability

O1: using private pointer instead of shared pointer
O2: using structure copy instead of element by element

Performance of Optimized UPC versus C + MPI for Matrix Multiplication



a. Execution time



b. Scalability

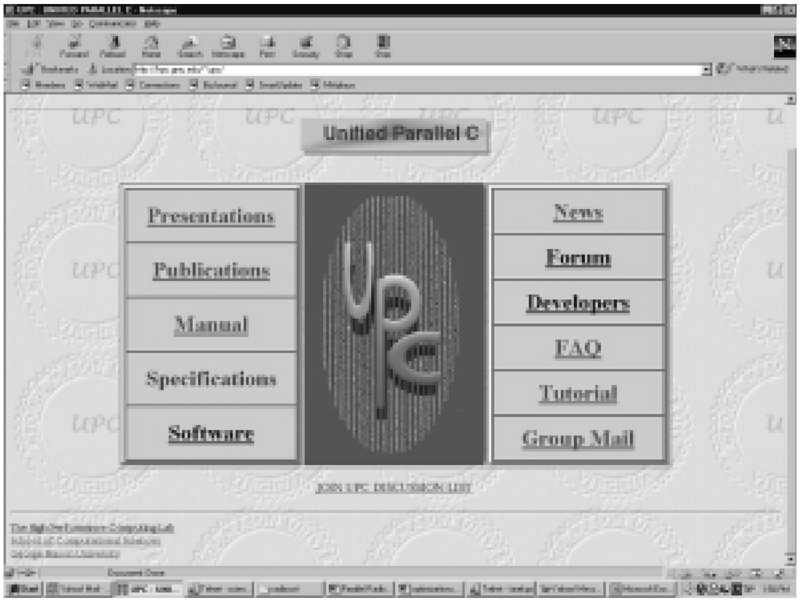
UPC Outline

1. Background and Philosophy
2. UPC Execution Model
3. UPC Memory Model
4. UPC: A Quick Intro
5. Data, Pointers, and Work Sharing
6. Dynamic Memory Management
7. Programming Examples
8. Synchronization
9. Performance Tuning and Early Results
10. Concluding Remarks

Conclusions

- UPC is easy to program in for C writers, significantly easier than alternative paradigms at times
- UPC exhibits very little overhead when compared with MPI for problems that are embarrassingly parallel. No tuning is necessary.
- For other problems compiler optimizations are happening but not fully there
- With hand-tuning, UPC performance compared favorably with MPI on the Compaq AlphaServer
- Hand tuned code, with block moves, is still substantially simpler than message passing code

http://upc.gwu.edu



A Co-Array Fortran Tutorial

Robert W. Numrich
Cray Inc.

Outline

1. Philosophy of Co-Array Fortran
2. Co-arrays and co-dimensions
3. Execution model
4. Relative image indices
5. Synchronization
6. Dynamic memory management
7. Example from UK Met Office
8. Examples from Linear Algebra
9. Using “Object-Oriented” Techniques with Co-Array Fortran
10. I/O
11. Summary

1. The Co-Array Fortran Philosophy

The Co-Array Fortran Philosophy

- What is the smallest change required to make Fortran 90 an effective parallel language?
- How can this change be expressed so that it is intuitive and natural for Fortran programmers to understand?
- How can it be expressed so that existing compiler technology can implement it efficiently?

The Co-Array Fortran Standard

- Co-Array Fortran is defined by:
  - R.W. Numrich and J.K. Reid, “Co-Array Fortran for Parallel Programming”, ACM Fortran Forum, 17(2):1-31, 1998
- Additional information on the web:
  - www.co-array.org

Co-Array Fortran on the T3E

- CAF has been a supported feature of Fortran 90 since release 3.1
- f90 -Z src.f90
- mpprun -n7 a.out

Non-Aligned Variables in SPMD Programs

- Addresses of arrays are on the local heap.
- Sizes and shapes are different on different program images.
- One processor knows nothing about another’s memory layout.
- How can we exchange data between such non-aligned variables?

Some Solutions

- **MPI-1**
  - Elaborate system of buffers
  - Two-sided send/receive protocol
  - Programmer moves data between local buffers only.
- **SHMEM**
  - One-sided exchange between variables in COMMON
  - Programmer manages non-aligned addresses and computes offsets into arrays to compensate for different sizes and shapes
- **MPI-2**
  - Mimic SHMEM by exposing some of the buffer system
  - One-sided data exchange within predefined windows
  - Programmer manages addresses and offsets within the windows

Co-Array Fortran Solution

- Incorporate the SPMD Model into Fortran 95 itself
  - Mark variables with co-dimensions
  - Co-dimensions behave like normal dimensions
  - Co-dimensions match problem decomposition not necessarily hardware decomposition
- The underlying run-time system maps your problem decomposition onto specific hardware.
- One-sided data exchange between co-arrays
  - Compiler manages remote addresses, shapes and sizes

The CAF Programming Model

- Multiple images of the same program (SPMD)
  - Replicated text and data
  - The program is written in a sequential language.
  - An “object” has the same name in each image.
  - Extensions allow the programmer to point from an object in one image to the same object in another image.
  - The underlying run-time support system maintains a map among objects in different images.

2. Co-Arrays and Co-Dimensions

What is Co-Array Fortran?

- Co-Array Fortran (CAF) is a simple parallel extension to Fortran 90/95.
- It uses normal rounded brackets ( ) to point to data in local memory.
- It uses square brackets [ ] to point to data in remote memory.
- Syntactic and semantic rules apply separately but equally to ( ) and [ ].

What Do Co-dimensions Mean?

The declaration

```
real :: x(n)[p,q,*]
```

means

1. An array of length n is replicated across images.
2. The underlying system must build a map among these arrays.
3. The logical coordinate system for images is a three dimensional grid of size
4. (p,q,r) where r=num_images()/(pq)

Examples of Co-Array Declarations

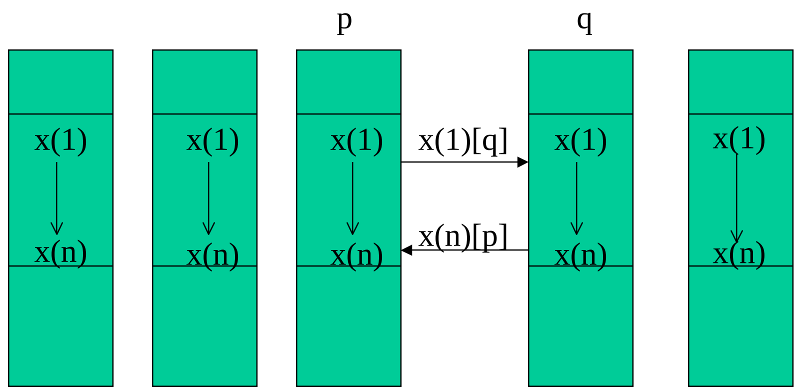
```
real :: a(n)[*]

real ::b(n)[p,*]

real ::c(n,m)[p,q,*]

complex,dimension[*] :: z

integer,dimension(n)[*] :: index

real,allocatable,dimension(:)[:] :: w

type(field), allocatable,dimension[:,:] :: maxwell
```

Communicating Between Co-Array "Objects"

```
y(:) = x(:)[p]
myIndex(:) = index(:)
yourIndex(:) = index(:)[you]
yourField = maxwell[you]
x(:)[q] = x(:) + x(:)[p]
x(index(:)) = y[index(:)]
```

Absent co-dimension defaults to the local object.

CAF Memory Model



Example I: A PIC Code Fragment

```
type(Pstruct) particle(myMax),buffer(myMax)[*]
myCell = this_image(buffer)
yours = 0
do mine =1,myParticles
    If(particle(mine)%x > rightEdge) then
        yours = yours + 1
        buffer(yours)[myCell+1] = particle( mine)
    endif
enddo
```

Exercise: PIC Fragment

- Convince yourself that no synchronization is required for this one-dimensional problem.
- What kind of synchronization is required for the three-dimensional case?
- What are the tradeoffs between synchronization and memory usage?

3. Execution Model

The Execution Model (I)

- The number of images is fixed.
- This number can be retrieved at run-time.

num_images() >= 1

- Each image has its own index.
- This index can be retrieved at run-time.

1 <= this_image() <= num_images()

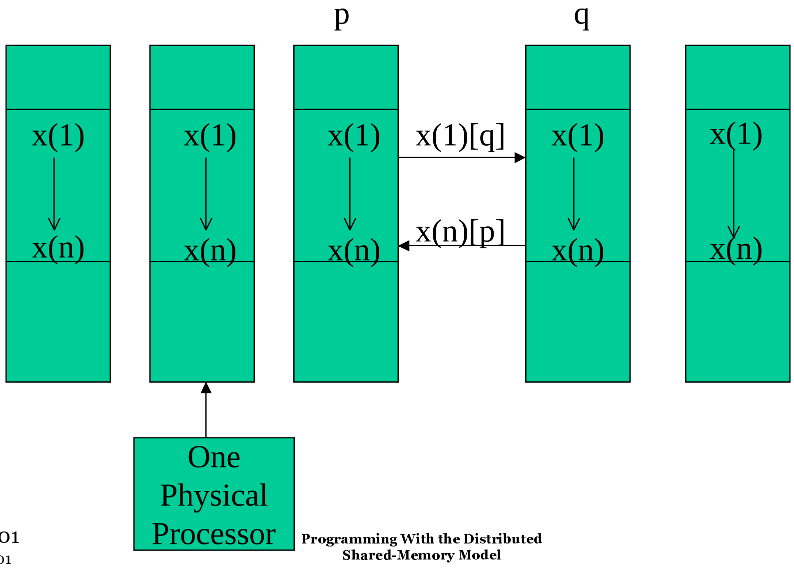
The Execution Model (II)

- Each image executes independently of the others.
- Communication between images takes place only through the use of explicit CAF syntax.
- The programmer inserts explicit synchronization as needed.

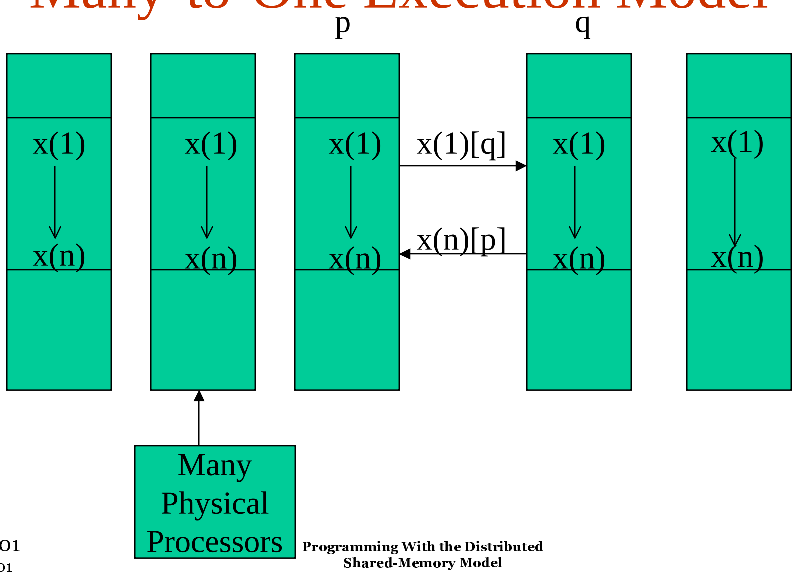
Who Builds the Map?

- The programmer specifies a **logical** map using co-array syntax.
- The underlying run-time system builds the **logical-to-virtual** map and a **virtual-to-physical** map.
- The programmer should be concerned with the logical map only.

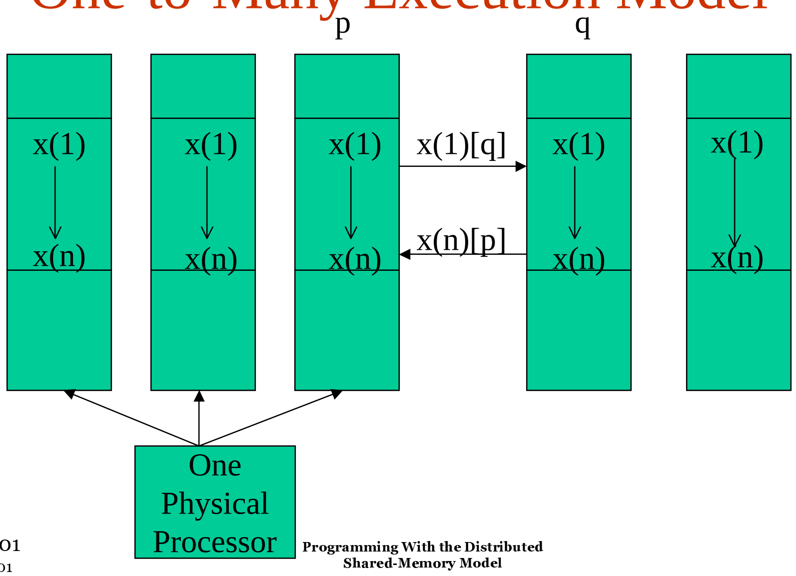
One-to-One Execution Model



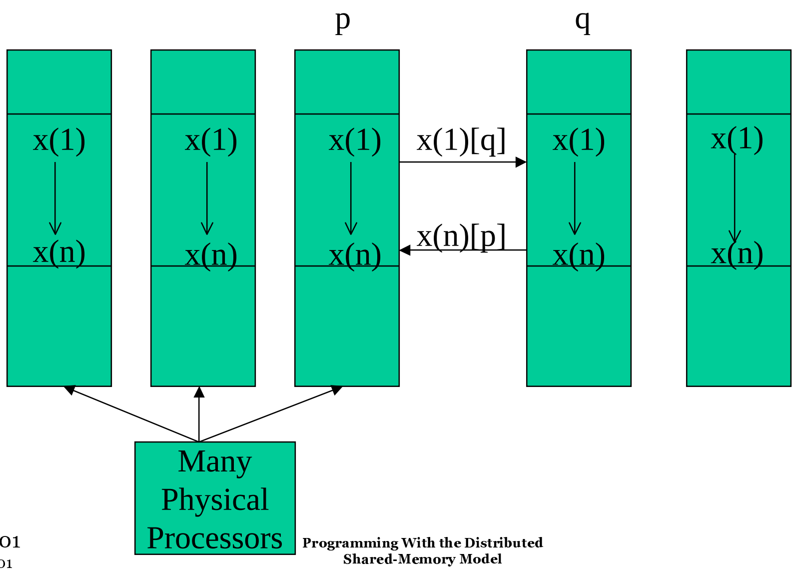
Many-to-One Execution Model



One-to-Many Execution Model



Many-to-Many Execution Model



---

4. Relative Image Indices

Relative Image Indices

- Runtime system builds a map among images.
- CAF syntax is a *logical* expression of this map.
- Current image index:

  1 <= this_image() <= num_images()
- Current image index relative to a co-array**:**

  lowCoBnd(x) <= this_image(x) <= upCoBnd(x)

Relative Image Indices (1)

	1	2	3	4
1	1	5	9	13
2	2	6	10	14
3	3	7	11	15
4	4	8	12	16

x[4,*]   this_image() = 15   this_image(x) = (/3,4/)

Relative Image Indices (II)

	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

x[0:3,0:*] this_image() = 15 this_image(x) = (/2,3/)

Relative Image Indices (III)

	0	1	2	3
-5	1	5	9	13
-4	2	6	10	14
-3	3	7	11	15
-2	4	8	12	16

x[-5:-2,0:*] this_image() = 15 this_image(x) = (/-3, 3/)

Relative Image Indices (IV)

	0	1	2	3	4	5	6	7
0	1	3	5	7	9	11	13	15
1	2	4	6	8	10	12	14	16

x[0:1,0:*]   this_image() = 15   this_image(x) =(/0,7/)

---

5. Synchronization

Synchronization Intrinsic Procedures

sync_all()

Full barrier; wait for all images before continuing.

sync_all(wait(:))

Partial barrier; wait only for those images in the wait(:) list.

sync_team(list(:))

Team barrier; only images in list(:) are involved.

sync_team(list(:),wait(:))

Team barrier; wait only for those images in the wait(:) list.

sync_team(myPartner)

Synchronize with one other image.

Events

sync_team(list(:),list(me:me)) post event

sync_team(list(:),list(you:you)) wait event

Example: Global Reduction

```
subroutine glb_dsum(x,n)
real(kind=8),dimension(n)[0:*] :: x
real(kind=8),dimension(n)   :: wrk
integer n,bit,i, mypartner,dim, me,  m
dim = log2_images()
if(dim .eq. 0) return
m  = 2**dim
bit = 1
me  = this_image(x)
do i=1,dim
  mypartner=xor(me,bit)
  bit=shiftl(bit,1)
  call sync_all()
  wrk(:) = x(:)[mypartner]
  call sync_all()
  x(:)=x(:)+wrk(:)
enddo
end subroutine glb_dsum
```

Exercise: Global Reduction

- Convince yourself that two sync points are required.
- How would you modify the routine to handle non-power-of-two number of images?
- Can you rewrite the example using only one barrier?

Other CAF Intrinsic Procedures

sync_memory()

Make co-arrays visible to all images

sync_file(unit)

Make local I/O operations visible to the global file system.

start_critical()

end_critical()

Allow only one image at a time into a protected region.

Other CAF Intrinsic Procedures

log2_images()

Log base 2 of the greatest power of two less than or equal to the value of num_images()

rem_images()

The difference between num_images() and the nearest power-of-two.

7. Dynamic Memory Management

Dynamic Memory Management

- Co-Arrays can be (should be) declared as allocatable

  real,allocatable,dimension(:,:)[:,:] :: x
- Co-dimensions are set at run-time

  allocate(x(n,n)[p,*])   implied sync
- Pointers are not allowed to be co-arrays

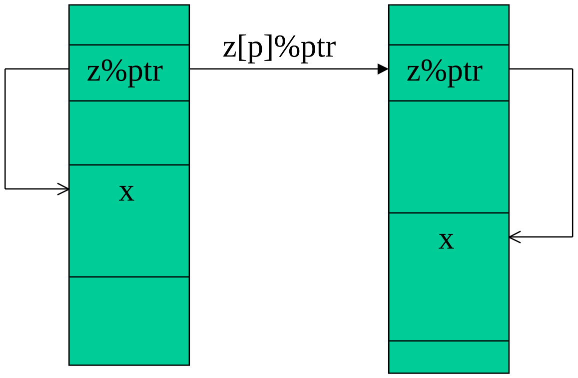
User Defined Derived Types

- F90 Derived types are similar to structures in C

```
type vector
   real, pointer,dimension(:) :: elements
   integer :: size
end type vector
```

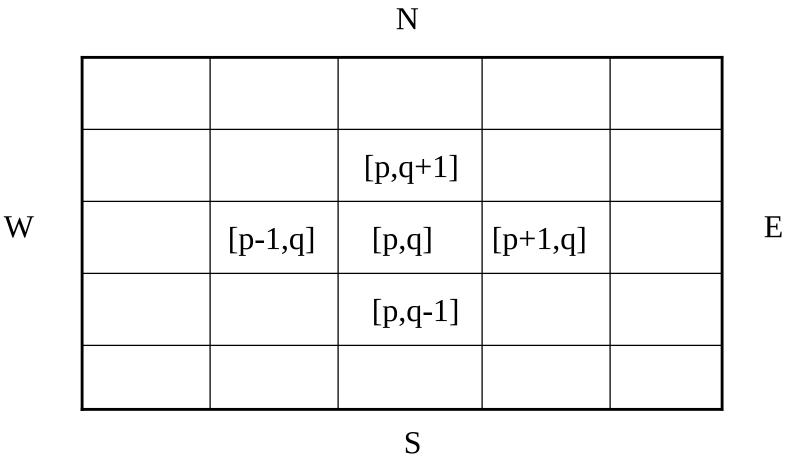
- Pointer components are allowed
- Allocatable components will be allowed in F2000

Irregular and Changing Data Structures



8. An Example from the UK Met Office

Problem Decomposition and Co-Dimensions



N

W

[p,q+1]

[p-1,q]

[p,q]

[p+1,q]

[p,q-1]

E

S

Cyclic Boundary Conditions in East-West Directions

```
myP = this_image(z,1)              !East-West

West = myP - 1
if(West < 1) West = nProcX          !Cyclic

East = myP + 1
if(East > nProcX) East = 1          !Cyclic
```

Incremental Update to Fortran 95

- Field arrays are allocated on the local heap.
- Define one supplemental F95 structure

```
type cafField
  real,pointer,dimension(:,:,:) :: Field
end type cafField
```

- Declare a co-array of this type

```
type(cafField),allocatable,dimension[:,:] :: z
```

Allocate Co-Array Structure

allocate (z [nP,*])

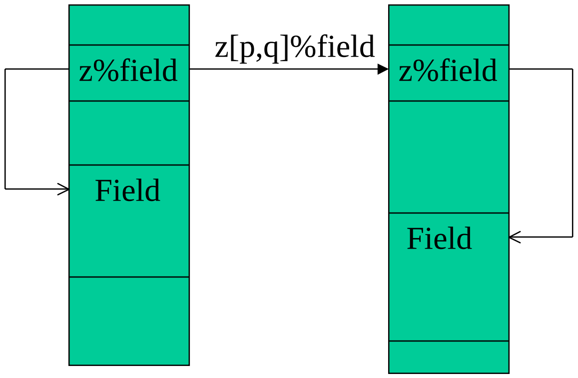
- Implied synchronization
- Structure is aligned across memory images.
  - Every image knows how to find the pointer component in any other image.
- Set the co-dimensions to match your problem decomposition

Local Alias to Remote Data

z%Field => Field

- Pointer assignment creates an alias to the local Field.
- The local Field is not aligned across memory images.
- But the alias is aligned because it is a component of an aligned co-array.

Co-Array Alias to a Remote Field



East-West Communication

- Move last row from west to my first halo
  - **Field(0,1:n,:) = z [West, myQ]%Field(m,1:n,:)**
- Move first row from east to my last halo
  - **Field(m+1,1:n,:) = z [East, myQ]%Field(1,1:n,:)**

Total Time (s)

PxQ	SHMEM	SHMEM w/CAF SWAP	MPI w/CAF SWAP	MPI
2x2	191	198	201	205
2x4	95.0	99.0	100	105
2x8	49.8	52.2	52.7	55.5
4x4	50.0	53.7	54.4	55.9
4x8	27.3	29.8	31.6	32.4

Other Kinds of Communication

- Semi-Lagrangian on-demand lists
  Field(i,list1(:),k) =z [myPal]% Field(i,list2(:),k)

- Gather data from a list of neighbors
  Field(i, j,k) = z [list(:)]%Field(i,j,k)

- Combine arithmetic with communication
  Field(i, j,k) = scale*z [myPal]%Field(i,j,k)

6. Examples from Linear Algebra

Blocked Matrices (1)

```
type matrix
   real,pointer,dimension(:,:) :: elements
   integer :: rowSize, colSize
end type matrix

type blockMatrix
   type(matrix),pointer,dimension(:,:) :: block
end type blockMatrix
```

Blocked Matrices (2)

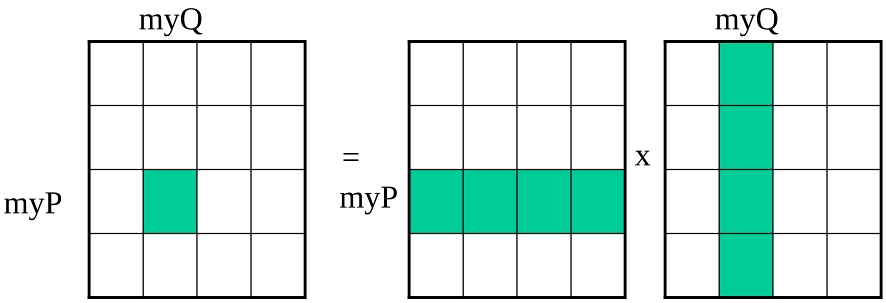
```
type(blockMatrix),allocatable :: a[:,:]
allocate(a[p,*])
allocate(a%block(nRowBlks,nColBlks))
a%block(j,k)%rowSize = nRows
a%block(j,k)%colSize = nCols
```

Irregular and Changing Data Structures

- Co-arrays of derived type vectors can be used to create sparse matrix structures.

```
type(vector),allocatable,dimension(:)[:] :: rowMatrix
allocate(rowMatrix(n)[*])
do i=1,n
   m = rowSize(i)
   rowMatrix(i)%size = m
   allocate(rowMatrix(i)%elements(m))
enddo
```

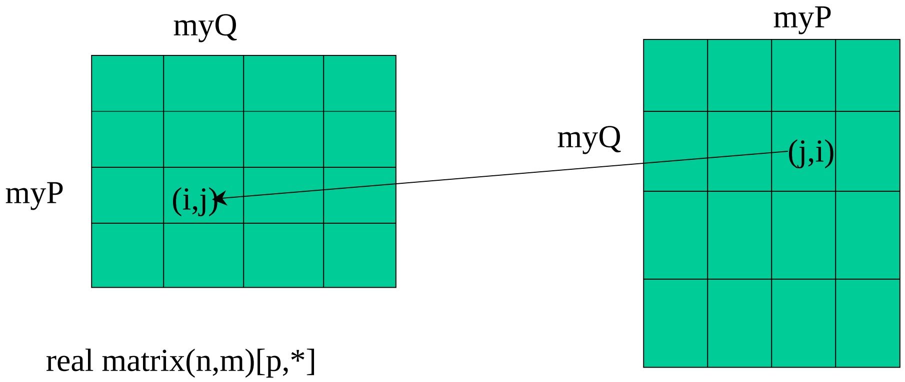
Matrix Multiplication



Matrix Multiplication

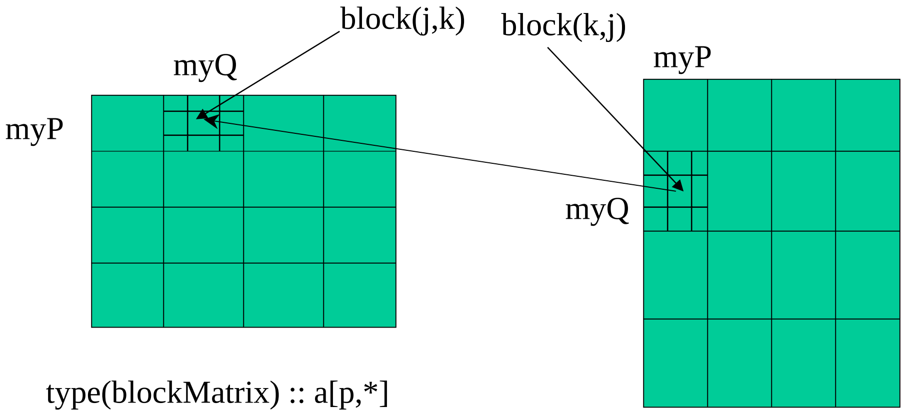
```
real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
do q=1,num_images()/p
    c(i,j) = c(i,j) + a(i,k)[myP, q]*b(k,j)[q,myQ]
enddo
enddo
```

Distributed Transpose (1)



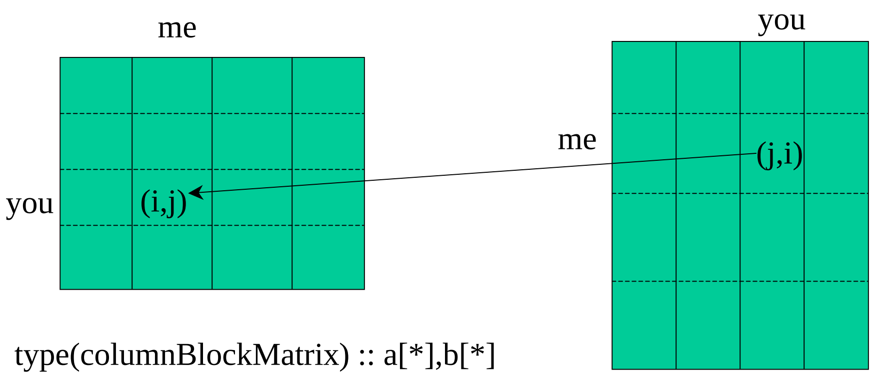
```
real matrix(n,m)[p,*]
matrix[myP,myQ](i,j) = matrix(j,i)[myQ,myP]
```

Distributed Transpose (2)



```
type(blockMatrix) :: a[p,*]
a%block(j,k)%element(i,j) = a[myQ,myP]%block(k,j)%elemnt(j,i)
```

Distributed Transpose (3)



type(columnBlockMatrix) :: a[*],b[*]
a[me]%block(you)%element(i,j) = b[you]%block(me)%element(j,i)

---

9. Using “Object-Oriented” Techniques with Co-Array Fortran

Using “Object-Oriented” Techniques with Co-Array Fortran

- Fortran 95 is not an object-oriented language.
- It contains some features that can be used to emulate object-oriented programming methods.
  - Named derived types are similar to classes without methods.
  - Modules can be used to associate methods loosely with objects.
  - Generic interfaces can be used to overload procedures based on the named types of the actual arguments.

CAF Parallel “Class Libraries”

```
program main
use blockMatrices
type(blockMatrix) :: x
type(blockMatrix) :: y[*]
call new(x)
call new(y)
call luDecomp(x)
call luDecomp(y)
end program main
```

9. CAF I/O

CAF I/O (1)

- There is one file system visible to all images.
- An image can open a file alone or as part of a team.
- The programmer controls access to the file using direct access I/O and CAF intrinsic functions.

CAF I/O (2)

- A new keyword , **team= ,** has been added to the open statement:
  - open(unit=,file=,team=list,access=direct)
  - Implied synchronization among team members.
- A CAF intrinsic function is provided to control file consistency across images:
  - call sync_file(unit)
  - Flush all local I/O operations to make them visible to the global file system.

CAF I/O (3)

- Read from unit 10 and place data in x(:) on image p.

  read(10,*) x(:)[p]
- Copy data from x(:) on image p to a local buffer and then write it to unit 10.

  write(10,*) x(:)[p]
- Write to a specified record in a file:

  write(unit,rec=myPart) x(:)[q]

10. Summary

Why Language Extensions?

- Languages are truly portable.
- There is no need to define a new language.
- Syntax gives the programmer control and flexibility
- Compiler concentrates on local code optimization.

Why Language Extensions?

- Compiler evolves as the hardware evolves.
  - Lowest latency allowed by the hardware.
  - Highest bandwidth allowed by the hardware.
  - Data ends up in registers or cache not in memory
  - Arbitrary communication patterns
  - Communication along multiple channels

Summary

- Co-dimensions match your problem decomposition
  - Run-time system matches them to hardware decomposition
  - Local computation of neighbor relationships
  - Flexible communication patterns
- Code simplicity
  - Non-intrusive code conversion
  - Modernize code to Fortran 95 standard
- Performance is comparable to or better than library based models.

Titanium: A Java Dialect for High Performance Computing

Kathy Yelick

U.C. Berkeley
Computer Science Division
http://www.cs.berkeley.edu/projects/titanium

Titanium Group

- Susan Graham
- Katherine Yelick
- Paul Hilfinger
- Phillip Colella (LBNL)
- Alex Aiken

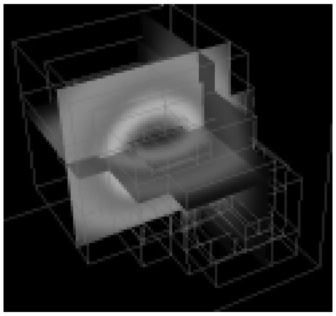
- Greg Balls (SDSC)
- Peter McQuorquodale (LBNL)

- Andrew Begel
- Dan Bonachea
- Tyson Condie
- David Gay
- Ben Liblit
- Chang Sun Lin
- Geoff Pike
- Jimmy Su
- Siu Man Yau

Target Problems

- **Many modeling problems in astrophysics, biology, material science, and other areas require**
  - Enormous range of spatial and temporal scales
- **To solve interesting problems, one needs:**
  - Adaptive methods
  - Large scale parallel machines
- **Titanium is designed for methods with**
  - Stuctured grids
  - Locally-structured grids (AMR)

Common Requirements



- **Algorithms for numerical PDE computations are**
  - communication intensive
  - memory intensive
- **AMR makes these harder**
  - more small messages
  - more complex data structures
  - most of the programming effort is debugging the boundary cases
  - locality and load balance trade-off is hard

Why Java for Scientific Computing?

- **Computational scientists use increasingly complex models**
  - Popularized C++ features: classes, overloading, pointer-based data structures
- **But C++ is very complicated**
  - easy to lose performance and readability
- **Java is a better C++**
  - Safe: strongly typed, garbage collected
  - Much simpler to implement (research vehicle)
  - May use the language without the JVM model

Summary of Features Added to Java

- **Multidimensional arrays with iterators**
- **Immutable (“value”) classes**
- **Templates**
- **Operator overloading**
- **Scalable SPMD parallelism**
- **Global address space**
- **Checked Synchronization**
- **Zone-based memory management**
- **Scientific Libraries**

Outline

- Titanium Execution Model
  - SPMD
  - Global Synchronization
  - Single
- Titanium Memory Model
- Support for Serial Programming
- Performance and Applications

SPMD Execution Model

- **Titanium has the same execution model as UPC and CAF.**
- **Basic Java programs may be run as Titanium, but all processors do all the work.**
- **E.g., parallel hello world**

```
class HelloWorld {
  public static void main (String [] argv) {
    System.out.println(“Hello from proc “ +
                       Ti.thisProc());
    }
 }
```

- **Any non-trivial program will have communication and synchronization**

SPMD Model

- All processors start together and execute same code, but not in lock-step
- Basic control done using
  - Ti.numProcs() total number of processors
  - Ti.thisProc() number of executing processor
- Bulk-synchronous style

```
read all particles and compute forces on mine
 Ti.barrier();
 write to my particles using new forces
 Ti.barrier();
```

- This is neither message passing nor data-parallel

Barriers and Single

- Common source of bugs is barriers or other global operations inside branches or loops

```
barrier, broadcast, reduction, exchange
```

- A “single” method is one called by all procs

```
public single static void allStep(...)
```

- A “single” variable has same value on all procs

```
int single timestep = 0;
```

- Single annotation on methods (also called “sglobal”) is optional, but useful to understanding compiler messages.

Explicit Communication: Broadcast

- **Broadcast is a one-to-all communication**

```
broadcast <value> from <processor>
```

- **For example:**

```
int count = 0;
int allCount = 0;
if (Ti.thisProc() == 0) count = computeCount();
allCount = broadcast count from 0;
```

- **The processor number in the broadcast must be single; all constants are single.**
  - **All processors must agree on the broadcast source.**
- **The allCount variable could be declared single.**
  - **All processors will have the same value after the broadcast.**

Example of Data Input

- **Same example, but reading from keyboard**
- **Shows use of Java exceptions**

```
int myCount = 0;
int single allCount = 0;
if (Ti.thisProc() == 0)
  try {
     DataInputStream kb = new
           DataInputStream(System.in);
     myCount =
         Integer.valueOf(kb.readLine()).intValue();
  } catch (Exception e) {
    System.err.println("Illegal Input");
 }
allCount = broadcast myCount from 0;
```

More on Single

- **Global synchronization needs to be controlled**

  if (this processor owns some data) {
      compute on it
      barrier
  }

- **Hence the use of “single” variables in Titanium**
- **If a conditional or loop block contains a barrier, all processors must execute it**
  - conditions in such loops, if statements, etc. must contain only single variables

Single Variable Example

- **Barriers and single in N-body Simulation**

```
class ParticleSim {
    public static void main (String [] argv) {
    int single allTimestep = 0;
    int single allEndTime = 100;
    for (; allTimestep < allEndTime; allTimestep++){
       read all particles and compute forces on mine
       Ti.barrier();
       write to my particles using new forces
       Ti.barrier();
     }
   }
 }
```

- **Single methods inferred by the compiler**

Outline

- **Titanium Execution Model**
- **Titanium Memory Model**
  - – Global and Local References
  - – Exchange: Building Distributed Data Structures
  - – Region-Based Memory Management
- **Support for Serial Programming**
- **Performance and Applications**

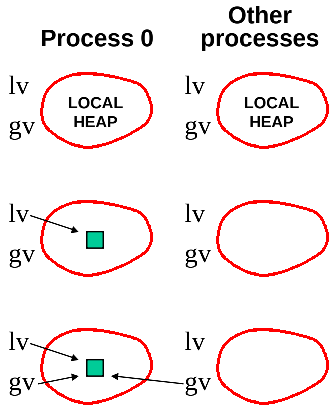
Use of Global / Local

- **As seen, references (pointers) may be remote**
  - – easy to port shared-memory programs
- **Global pointers are more expensive than local**
  - – True even when data is on the same processor
  - – Use local declarations in critical sections
- **Costs of global:**
  - – space (processor number + memory address)
  - – dereference time (check to see if local)
- **May declare references as local**
  - – Compiler will automatically infer them when possible

Global Address Space

- **Processes allocate locally**
- **References can be passed to other processes**

```
Class C { int val;... }
C gv;       // global pointer
C local lv; // local pointer

if (thisProc() == 0) {
      lv = new C();
}
gv = broadcast lv from 0;
gv.val = ...;
... = gv.val;
```



Shared/Private vs Global/Local

- **Titanium's global address space is based on pointers rather than shared variables**
- **There is no distinction between a private and shared heap for storing objects**
- **All objects may be referenced by global pointers or by local ones**
- **There is no direct support for distributed arrays**
  - Irregular problems do not map easily to distributed arrays, since each processor will own a set of objects (sub-grids)
  - For regular problems, Titanium uses pointer dereference instead of index calculation
  - Important to have local "views" of data structures

Aside on Titanium Arrays

- **Titanium adds its own multidimensional array class for performance**
- **Distributed data structures are built using a 1D Titanium array**
- **Slightly different syntax, since Java arrays still exist in Titanium, e.g.:**

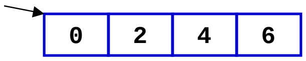
```
int [1d] arr;
arr = new int [100];
arr[1] = 4*arr[1];
```

Explicit Communication: Exchange

- **To create shared data structures**
  - each processor builds its own piece
  - pieces are exchanged (for object, just exchange pointers)
- **Exchange primitive in Titanium**

```
int [1d] single allData;
allData = new int [0:Ti.numProcs()-1];
allData.exchange(Ti.thisProc()*2);
```

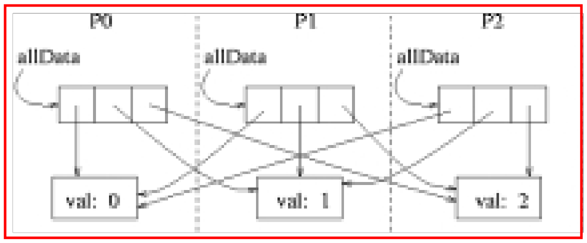
- **E.g., on 4 procs, each will have copy of allData:**



Building Distributed Structures

- **Distributed structures are built with exchange:**

```
class Boxed {
    public Boxed (int j) { val = j;}
    public int val;
 }
```



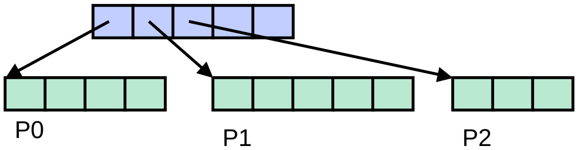
```
Object [1d] single allData;
allData = new Object [0:Ti.numProcs()-1];
allData.exchange(new Boxed(Ti.thisProc());
```

Distributed Data Structures

- **Building distributed arrays:**

```
Particle [1d] single [1d] allParticle =
                  new Particle [0:Ti.numProcs-1][1d];
Particle [1d] myParticle =
                  new Particle [0:myParticleCount-1];
allParticle.exchange(myParticle);
```

- **Now each processor has array of pointers, one to each processor's chunk of particles**



Region-Based Memory Management

- **An advantage of Java over C/C++ is:**
  - Automatic memory management
- **But garbage collection is:**
  - Has a reputation of slowing serial code
  - Is hard to implement and scale in a parallel environment
- **Titanium takes the following approach:**
  - Memory management is safe – cannot deallocate live data
  - Garbarge collection is as default (most platforms)
  - Higher performance is possible using regions

Region-Based Memory Management

- **Need to organize data structures**
- **Allocate set of objects (safely)**

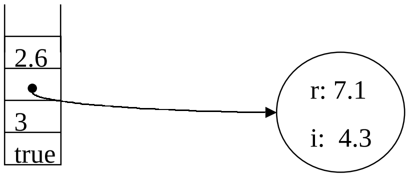
```
PrivateRegion r = new PrivateRegion();
for (int j = 0; j < 10; j++) {
   int[] x = new ( r ) int[j + 1];
   work(j, x);
}
try { r.delete(); }
catch (RegionInUse oops) {
    System.out.println(“failed to delete”);
  }
}
```

Outline

- **Titanium Execution Model**
- **Titanium Memory Model**
- **Support for Serial Programming**
  - **Immutables**
  - **Multidimensional arrays**
  - **Operator overloading**
  - **Templates**
- **Performance and Applications**

Java Objects

- **Primitive scalar types: boolean, double, int, etc.**
  - **implementations will store these on the program stack**
  - **access is fast -- comparable to other languages**
- **Objects: user-defined and standard library**
  - **passed by pointer value (object sharing) into functions**
  - **has level of indirection (pointer to) implicit**
  - **simple model, but inefficient for small objects**



Java Object Example

```
class Complex {
  private double real;
  private double imag;
  public Complex(double r, double i) {
                   real = r; imag = i; }
  public Complex add(Complex c) {
            return new Complex(c.real + real, c.imag + imag);
  public double getReal {return real; }
  public double getImag {return imag;}
}
```

```
Complex c = new Complex(7.1, 4.3);
c = c.add(c);
class VisComplex extends Complex { ... }
```

Immutable Classes in Titanium

- **For small objects, would sometimes prefer**
  - **to avoid level of indirection**
  - **pass by value (copying of entire object)**
  - **especially when immutable -- fields never modified**
    - **extends the idea of primitive values to user-defined values**
- **Titanium introduces immutable classes**
  - **all fields are final (implicitly)**
  - **cannot inherit from or be inherited by other classes**
  - **needs to have 0-argument constructor**
- **Note: considering allowing mutation in future**

Example of Immutable Classes

– **The immutable complex class nearly the same**

```
immutable class Complex {
    Complex () {real=0; imag=0; }
    ...
}
```

new keyword

Zero-argument constructor required

Rest unchanged. No assignment to fields outside of constructors.

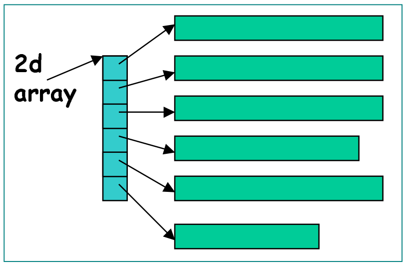
– **Use of immutable complex values**

```
Complex c1 = new Complex(7.1, 4.3);
Complex c2 = new Complex(2.5, 9.0);
c1 = c1.add(c2);
```

Similar to structs in C in terms of performance

Arrays in Java

- **Arrays in Java are objects**
- **Only 1D arrays are directly supported**
- **Multidimensional arrays are slow**



- **Subarrays are important in AMR (e.g., interior of a grid)**
  - **Even C and C++ don’t support these well**
  - **Hand-coding (array libraries) can confuse optimizer**

Multidimensional Arrays in Titanium

- **New multidimensional array added to Java**
  - One array may be a subarray of another
    - e.g., a is interior of b, or a is all even elements of b
  - Indexed by Points (tuples of ints)
  - Constructed over a set of Points, called Rectangular Domains (RectDomains)
  - Points, Domains and RectDomains are built-in immutable classes

- **Support for AMR and other grid computations**
  - domain operations: intersection, shrink, border

Unordered Iteration

- **Memory hierarchy optimizations are essential**
- **Compilers can sometimes do these, but hard in general**
- **Titanium adds unordered iteration on rectangular domains**

```
foreach (p in r) { ... }
```

  - **p is a Point**
  - **r is a RectDomain or Domain**
- **Foreach simplifies bounds checking as well**
- **Additional operations on domains to subset and xform**
- **Note: foreach is not a parallelism construct**

Point, RectDomain, Arrays in General

- **Points specified by a tuple of ints**

```
Point<2> lb = [1, 1];
Point<2> ub = [10, 20];
```

- **RectDomains given by 3 points:**
  - **lower bound, upper bound (and stride)**

```
RectDomain<2> r = [lb : ub];
```

- **Array declared by # dimensions and type**

```
double [2d] a;
```

- **Array created by passing RectDomain**

```
a = new double [r];
```

Simple Array Example

- **Matrix sum in Titanium**

```
Point<2> lb = [1,1];
Point<2> ub = [10,20];
RectDomain<2> r = [lb,ub];

double [2d] a = new double [r];
double [2d] b = new double [1:10,1:20];
double [2d] c = new double [lb:ub:[1,1]];

for (int i = 1; i <= 10; i++)
   for (int j = 1; j <= 20; j++)
     c[i,j] = a[i,j] + b[i,j];
```

No array allocation here

Syntactic sugar

Optional stride

Naïve MatMul with Titanium Arrays

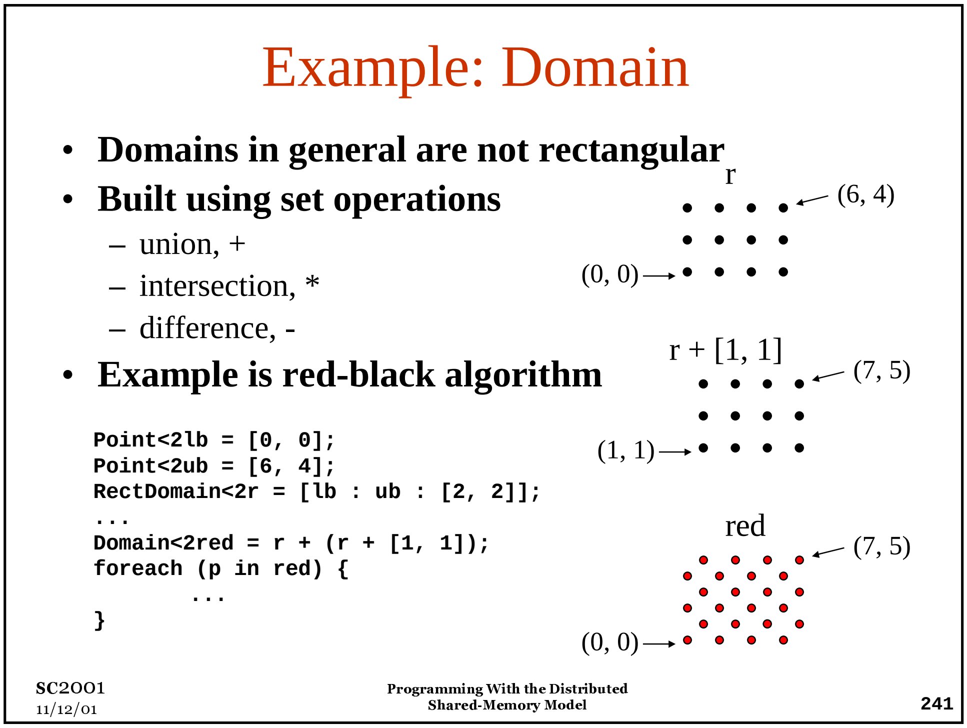
```
public static void matMul(double [2d] a, double [2d] b,
                          double [2d] c) {
  int n = c.domain().max()[1];  // assumes square
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
           c[i,j] += a[i,k] * b[k,j];
         }
      }
   }
}
```

Better MatMul with Titanium Arrays

```
public static void matMul(double [2d] a, double [2d] b,
                          double [2d] c) {
  foreach (ij within c.domain()) {
    double [1d] aRowi = a.slice(1, ij[1]);
    double [1d] bColj = b.slice(2, ij[2]);
    foreach (k within aRowi.domain()) {
      c[ij] += aRowi[k] * bColj[k];
    }
  }
}
```

Current performance: comparable to 3 nested loops in C

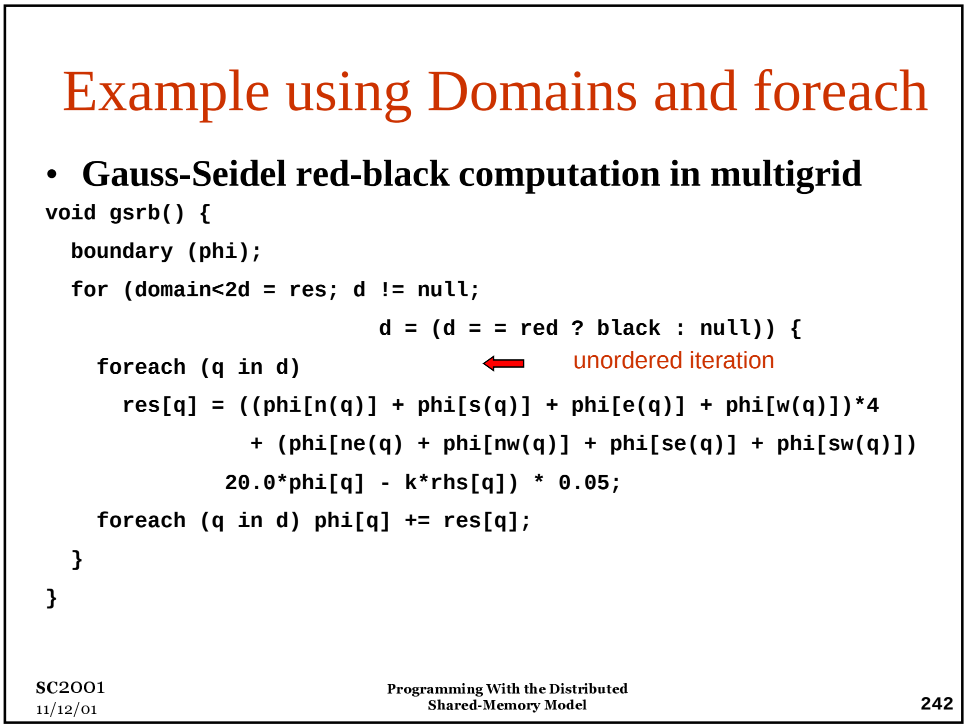
Future: automatic blocking for memory hierarchy (Geoff Pike’s PhD thesis)



Example: Domain

- **Domains in general are not rectangular**
- **Built using set operations**
  - – union, +
  - – intersection, *
  - – difference, -
- **Example is red-black algorithm**

```
Point<2lb = [0, 0];
Point<2ub = [6, 4];
RectDomain<2r = [lb : ub : [2, 2]];
...
Domain<2red = r + (r + [1, 1]);
foreach (p in red) {
        ...
}
```



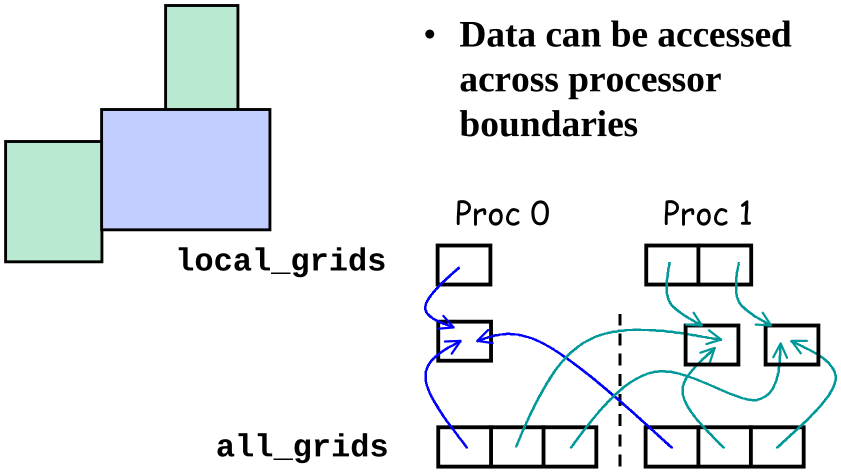
Example using Domains and foreach

- **Gauss-Seidel red-black computation in multigrid**

```
void gsrb() {
  boundary (phi);
  for (domain<2d = res; d != null;
                         d = (d = = red ? black : null)) {
    foreach (q in d)
      res[q] = ((phi[n(q)] + phi[s(q)] + phi[e(q)] + phi[w(q)])*4
                + (phi[ne(q) + phi[nw(q)] + phi[se(q)] + phi[sw(q)])
              20.0*phi[q] - k*rhs[q]) * 0.05;
    foreach (q in d) phi[q] += res[q];
  }
}
```

unordered iteration

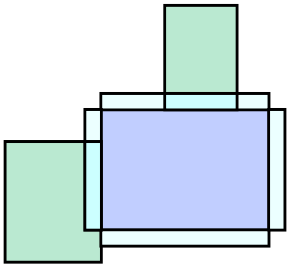
Example: A Distributed Data Structure



- **Data can be accessed across processor boundaries**

Example: Setting Boundary Conditions

```
foreach (l in local_grids.domain()) {
  foreach (a in all_grids.domain()) {
     local_grids[l].copy(all_grids[a]);
  }
}
```



Overloading in Titanium

- **For convenience, Titanium also provides overloading**

```
class Complex {
  private double real;
  private double imag;
  public Complex operator+(Complex c) {
            return new Complex(c.real + real,
                               c.imag + imag);
  ...
}
```

```
Complex c1 = new Complex(7.1, 4.3);
Complex c2 = new Complex(5.4, 3.9);
Complex c3 = c1 + c2;
```

Templates

- **Many applications use containers:**
  - E.g., arrays parameterized by dimensions, element types
  - Java supports this kind of parameterization through inheritance
    - Only put Object types into contains
    - Inefficient when used extensively
- **Titanium provides a template mechanism like C++**
  - Used to build a distributed array package
  - Hides the details of exchange, indirection within the data structure, etc.

Using Templates: Distributed Arrays

```
template <class T, int single arity> public class
  DistArray {
  RectDomain <arity> single rd;
  T [arity d][arity d] subMatrices;
  RectDomain <arity> [arity d] single subDomains;
  ...
 /* Sets the element at p to value */
  public void set (Point <arity> p, T value) {
    getHomingSubMatrix (p) [p] = value;
  }
}
```

```
template DistArray <double, 2> single A = new template
  DistArray <double, 2> ([ [0, 0] : [aHeight, aWidth]);
```

Outline

- Titanium Execution Model
- Titanium Memory Model
- Support for Serial Programming
- Performance and Applications
  - Serial Performance on pure Java (SciMark)
  - Parallel Applications
- Compiler Optimizations

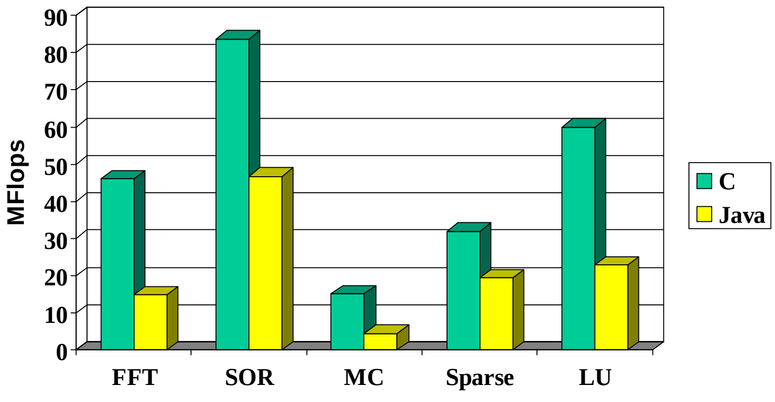
SciMark Benchmark

- **Numerical benchmark for Java, C/C++**
- **Five kernels:**
  - FFT (complex, 1D)
  - Successive Over-Relaxation (SOR)
  - Monte Carlo integration (MC)
  - Sparse matrix multiply
  - dense LU factorization
- **Results are reported in Mflops**
- **Download and run on your machine from:**
  - http://math.nist.gov/scimark2
  - C and Java sources also provided

Roldan Pozo, NIST, http://math.nist.gov/~Rpozo

SciMark: Java vs. C

(Sun UltraSPARC 60)

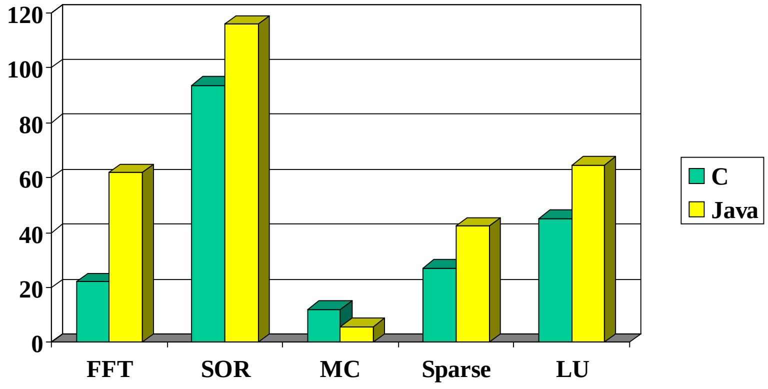


* Sun JDK 1.3 (HotSpot) , javac -0; Sun cc -0; SunOS 5.7

Roldan Pozo, NIST, http://math.nist.gov/~Rpozo

SciMark: Java vs. C

(Intel PIII 500MHz, Win98)



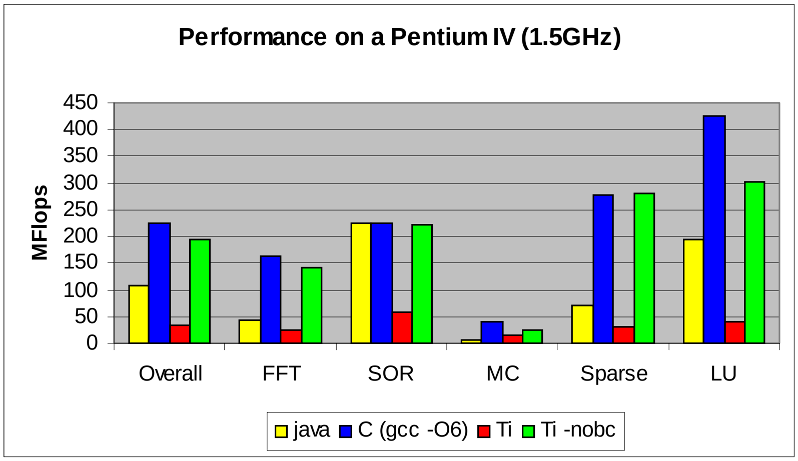
* Sun JDK 1.2, javac -0; Microsoft VC++ 5.0, cl -0; Win98

Roldan Pozo, NIST, http://math.nist.gov/~Rpozo

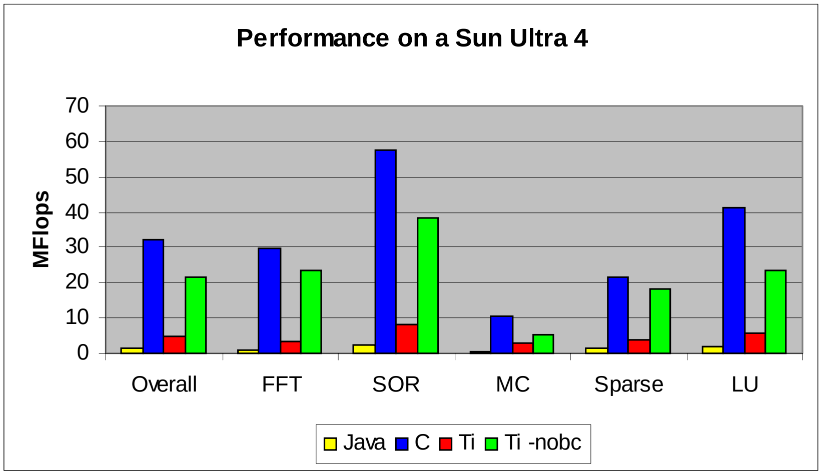
Can we do better without the JVM?

- **Pure Java with a JVM (and JIT)**
  - Within 2x of C and sometimes better
    - OK for many users, even those using high end machines
  - Depends on quality of both compilers
- **We can try to do better using a traditional compilation model**
  - E.g., Titanium compiler at Berkeley
    - Compiles Java extension to C
    - Does not optimize Java arrays or for loops (prototype)

Java Compiled by Titanium Compiler



Java Compiled by Titanium Compiler



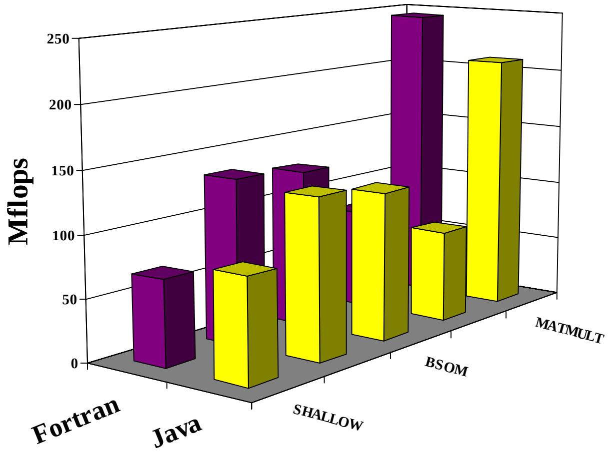
Language Support for Performance

- **Multidimensional arrays**
  - Contiguous storage
  - Support for sub-array operations without copying
- **Support for small objects**
  - E.g., complex numbers
  - Called "immutables" in Titanium
  - Sometimes called "value" classes
- **Unordered loop construct**
  - Programmer specifies iteration independent
  - Eliminates need for dependence analysis – short term solution? Used by vectorizing compilers.

HPJ Compiler from IBM

- **HPJ Compiler from IBM Research**
  - Moreira et. al
- **Program using Array classes which use contiguous storage**
  - e.g. A[i][j] becomes A.get(i,j)
  - No new syntax (worse for programming, but better portability – any Java compiler can be used)
- **Compiler for IBM machines, exploits hardware**
  - e.g., Fused Multiply-Add
- **Result: 85+% of Fortran on RS/6000**

Java vs. Fortran Performance



*IBM RS/6000 67MHz POWER2 (266 Mflops peak) AIX Fortran, HPJC

Array Performance Issues

- **Array representation is fast, but access methods can be slow, e.g., bounds checking, strides**
- **Compiler optimizes these**
  - common subexpression elimination
  - eliminate (or hoist) bounds checking
  - strength reduce: e.g., naïve code has 1 divide per dimension for each array access

- **Currently +/- 20% of C/Fortran for large loops**
- **Future: small loop and cache optimizations**

Parallel Applications

- **Genome Application**
- **Heart simulation**
- **AMR elliptic and hyperbolic solvers**
- **Scalable Poisson for infinite domains**
- **Genome application**
- **Several smaller benchmarks: EM3D, MatMul, LU, FFT, Join**

MOOSE Application

- **Problem: Microarray construction**
  - Used for genome experiments
  - Possible medical applications long-term
- **Microarray Optimal Oligo Selection Engine (MOOSE)**
  - A parallel engine for selecting the best oligonucleotide sequences for genetic microarray testing
  - Uses dynamic load balancing within Titanium

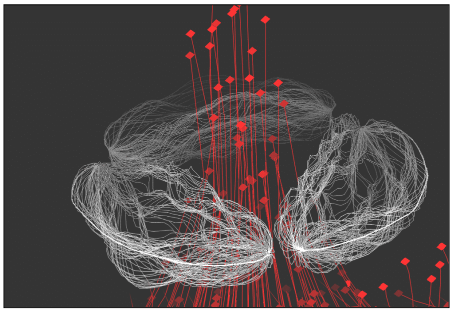
Heart Simulation

- **Problem: compute blood flow in the heart**
  - Modeled as an elastic structure in an incompressible fluid.
    - The “immersed boundary method” due to Peskin and McQueen.
    - 20 years of development in model
    - Many applications other than the heart: blood clotting, inner ear, paper making, embryo growth, and others
  - Use a regularly spaced mesh (set of points) for evaluating the fluid
- **Uses**
  - Current model can be used to design heart valves
  - Related projects look at the behavior of the heart during a heart attack
  - Ultimately: real-time clinical work

Heart Simulation Calculation

The involves solving Navier-Stokes equations

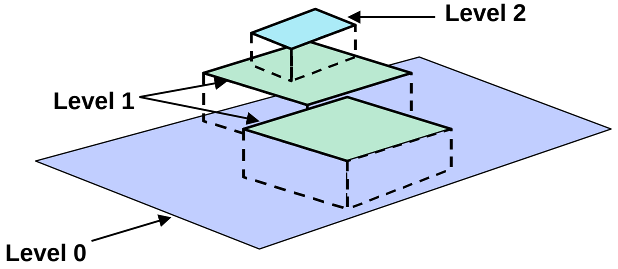
- –64^3 was possible on Cray YMP, but 128^3 required for accurate model (would have taken 3 years).
- –Done on a Cray C90 -- 100x faster and 100x more memory
- –Until recently, limited to vector machines



- Needs more features:
  - Electrical model of the heart, and details of muscles, E.g.,
    - Chris Johnson
    - Andrew McCulloch
  - Lungs, circulatory systems

AMR Poisson

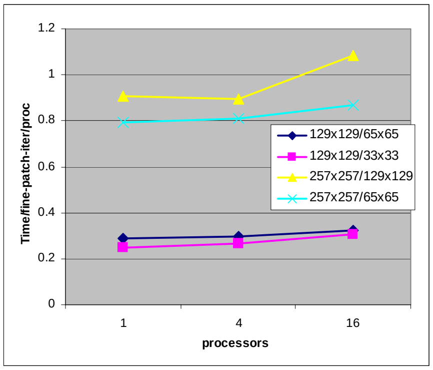
- **Poisson Solver [Semenzato, Pike, Colella]**
  - 3D AMR
  - finite domain
  - variable coefficients
  - multigrid across levels



- **Performance of Titanium implementation**
  - Sequential multigrid performance +/- 20% of Fortran
  - On fixed, well-balanced problem of 8 patches, each 72^3
  - parallel speedups of 5.5 on 8 processors

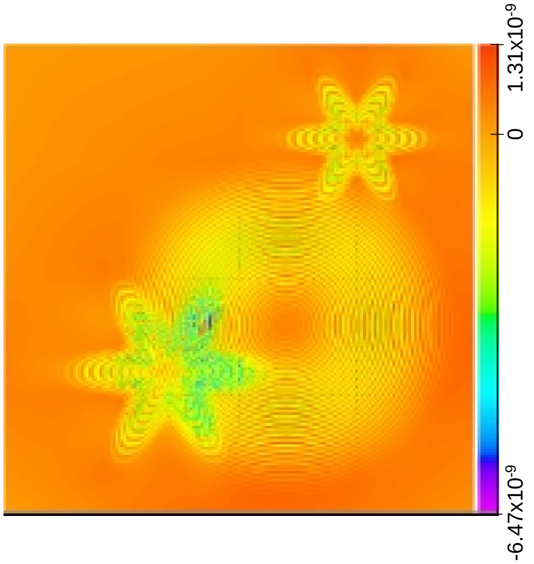
Scalable Poisson Solver

- **MLC for Finite-Differences by Balls and Colella**
- **Poisson equation with infinite boundaries**
  - arise in astrophysics, some biological systems, etc.
- **Method is scalable**
  - Low communication
- **Performance on**
  - SP2 (shown) and t3e
  - scaled speedups
  - nearly ideal (flat)
- **Currently 2D and non-adaptive**



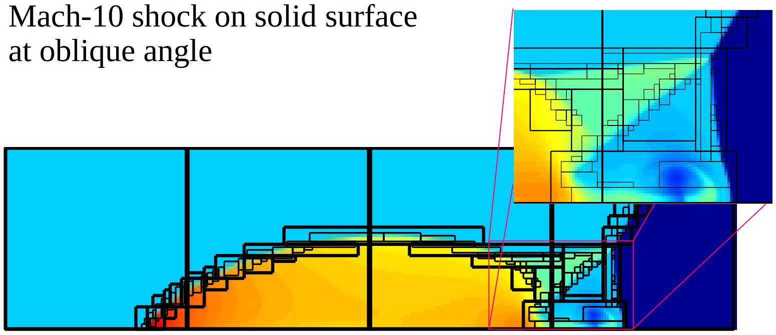
Error on High-Wavenumber Problem

- **Charge is**
  - 1 charge of concentric waves
  - 2 star-shaped charges.
- **Largest error is where the charge is changing rapidly.** Note:
  - discretization error
  - faint decomposition error
- **Run on 16 procs**



AMR Gas Dynamics

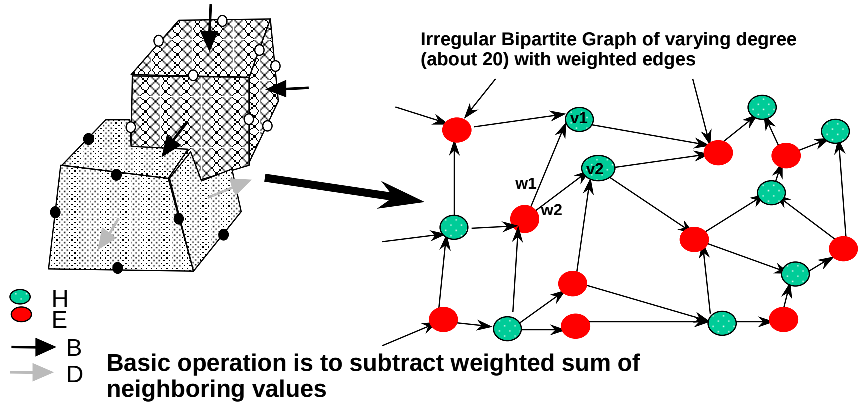
- **Developed by McCorquodale and Colella**
- **2D Example (3D supported)**
  - Mach-10 shock on solid surface at oblique angle



- **Future: Self-gravitating gas dynamics package**

An Irregular Problem: EM3D

Maxwells Equations on an Unstructured 3D Mesh: Explicit Method



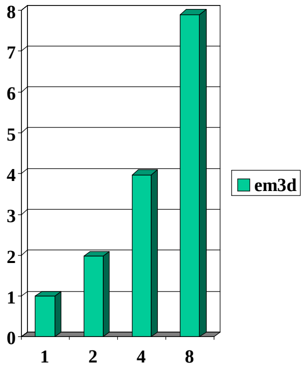
Irregular Bipartite Graph of varying degree (about 20) with weighted edges

Basic operation is to subtract weighted sum of neighboring values

for all E nodes
for all H nodes

Unstructured Mesh Kernel

- **EM3D: Relaxation on a 3D unstructured mesh**
- **Speedup on Ultrasparc SMP**
- **Simple kernel: mesh not partitioned.**



Calling Other Languages

- **We have built interfaces to**
  - PETSc : scientific library for finite element applications
  - Metis: graph partitioning library
  - KeLP: starting work on this
- **Two issues with cross-language calls**
  - accessing Titanium data structures (arrays) from C
    - possible because Titanium arrays have same format on inside
  - having a common message layer
    - Titanium is built on lightweight communication

Implementation Status

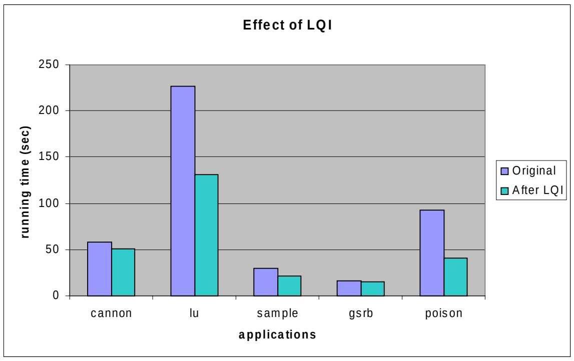
- **Strategy**
  - Titanium into C
  - Solaris or Posix threads for SMPs
  - Lightweight communication for MPPs/Clusters
    - Active messages, LAPI, shmem, MPI, UDP, others…
- **Status: Titanium runs on**
  - Solaris or Linux SMPs, clusters, CLUMPS
  - Berkeley NOW & Berkeley Millennium clusters
  - Cray T3E (NERSC and NPACI)
  - IBM SP2/SP Power3
  - SGI Origin 2000

Outline

- **Titanium Execution Model**
- **Titanium Memory Model**
- **Support for Serial Programming**
- **Performance and Applications**
- **Compiler Optimizations**
  - Local pointer identification (LQI)
  - Overlap of communication (Split-C experience)
  - Preserving the consistency model
    - Cycle detection: parallel dependence analysis
    - Synchronization analysis: parallel flow analysis

Local Pointer Analysis

- **Compiler can infer many uses of local**



- **Data structures must be well partitioned**

Split-C Experience: Latency Overlap

- **Titanium borrowed ideas from Split-C**
  - global address space
  - SPMD parallelism
- **But, Split-C had non-blocking accesses built in to tolerate network latency on remote read/write**

```
int *global p;
x := *p;   /* get */
*p := 3;   /* put */
sync;      /* wait for my puts/gets */
```

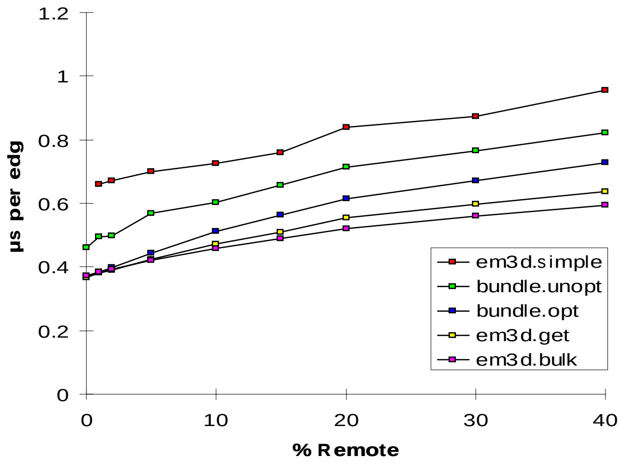
- **Also one-way communication**

```
*p :- x;         /* store */
all_store_sync;  /* wait globally */
```

- **Conclusion: useful, but complicated**

Split-C: Performance Tuning

- Tuning affects application performance



Consistency Model

- **Titanium adopts the Java memory consistency model**
- **Roughly: Access to shared variables that are not synchronized have undefined behavior.**
- **Use synchronization to control access to shared variables.**
  - barriers
  - synchronized methods and blocks

Parallel Optimizations

- **Two new analyses**
  - synchronization analysis: the parallel analog to control flow analysis for serial code [Gay & Aiken]
  - shared variable analysis: the parallel analog to dependence analysis [Krishnamurthy & Yelick]

Sources of Memory/Comm. Overlap

- **Would like compiler to introduce put/get/store.**
- **Hardware also reorders**
  - out-of-order execution
  - write buffered with read by-pass
  - non-FIFO write buffers
  - weak memory models in general
- **Software already reorders too**
  - register allocation
  - any code motion
- **System provides enforcement primitives**
  - e.g., memory fence, volatile, etc.
  - tend to be heavy wait and with unpredictable performance
- **Can the compiler hide all this?**

Semantics: Sequential Consistency

- **When compiling sequential programs:**

```
x = expr1;
y = expr2;
```

```
y = expr2;
x = expr1;
```

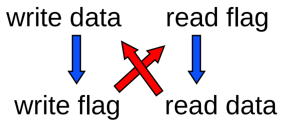
Valid if y not in expr1 and x not in expr2 (roughly)

- **When compiling parallel code, not sufficient test.**

```
Initially flag = data = 0
Proc A        Proc B
data = 1;     while (flag!=1);
flag = 1;     ... = ...data...;
```

Cycle Detection: Dependence Analog

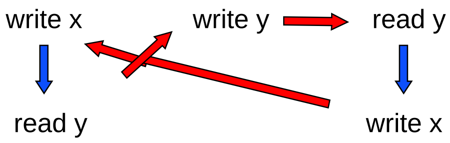
- **Processors define a “program order” on accesses from the same thread**
  - → **P is the union of these total orders**
- **Memory system define an “access order” on accesses to the same variable**
  - → **A is access order (read/write & write/write pairs)**



- **A violation of sequential consistency is cycle in P U A.**
- **Intuition: time cannot flow backwards.**

Cycle Detection

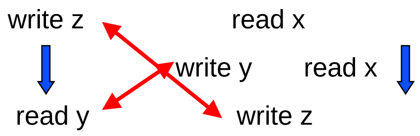
- **Generalizes to arbitrary numbers of variables and processors**



- **Cycles may be arbitrarily long, but it is sufficient to consider only cycles with 1 or 2 consecutive stops per processor**

Static Analysis for Cycle Detection

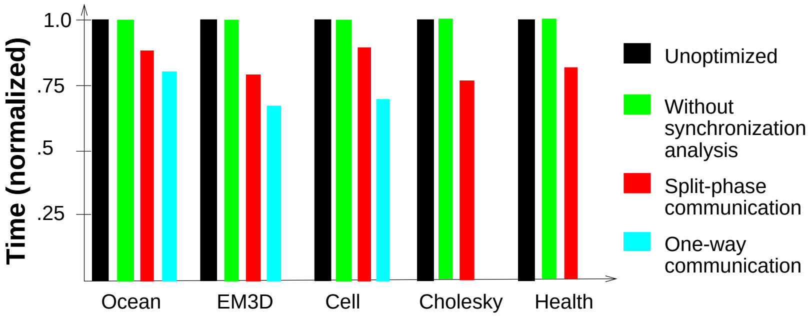
- **Approximate P by the control flow graph**
- **Approximate A by undirected "dependence" edges**
- **Let the "delay set" D be all edges from P that are part of a minimal cycle**


write z
read x
write y
read x
read y
write z

- **The execution order of D edge must be preserved; other P edges may be reordered (modulo usual rules about serial code)**
- **Synchronization analysis also critical [**

Communication Optimizations

- **Implemented in subset of C with limited pointers [Krishnamurthy, Yelick]**
- **Experiments on the NOW; 3 synchronization styles**



- **Future: pointer analysis and optimizations for AMR [Jeh, Yelick]**

Parallel Programming Using A Distributed Shared Memory Model

Summary

One Model

- Distributed Shared Memory
  - – Coding simplicity
  - – Recognizes system capabilities

Three Languages

- Small changes to existing languages
  - ANSI C ⇒ UPC
  - F90 ⇒ Co-Array Fortran
  - Java ⇒ Titanium
- Many implementations on the way

For More Info

- UPC
  - http://upc.gwu.edu
- Co-Array Fortran
  - http://www.co-array.org
- Titanium
  - http://www.cs.berkeley.edu/Research/Projects/titanium