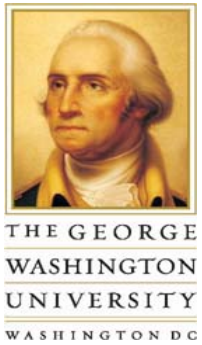


Berkeley
University of California

IDA

Programming in the Partitioned Global Address Space Model

Bill Carlson, IDA

Tarek El-Ghazawi, GWU

Robert Numrich, U. Minnesota

Kathy Yelick, UC Berkeley

Table of Contents

Introduction to the PGAS Model

Bill Carlson

IDA- Center for Computing Sciences

wwc@super.org

Naming Issues

- Focus of this tutorial
  - Partitioned Global Address Space (PGAS) Model, aka
  - Distributed Shared Memory Programming Model (DSM), aka
  - Locality Conscious Shared Space Model,
  - …

Outline of the Day

- Introduction to PGAS Model
- UPC Programming
- Co-Array Fortran Programming
- Titanium Programming
- Summary

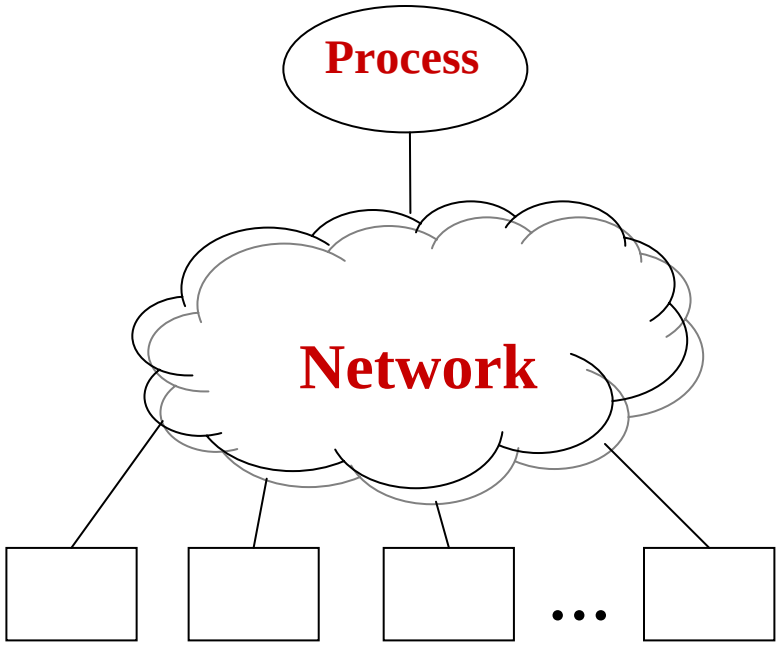
Outline of this Talk

- 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

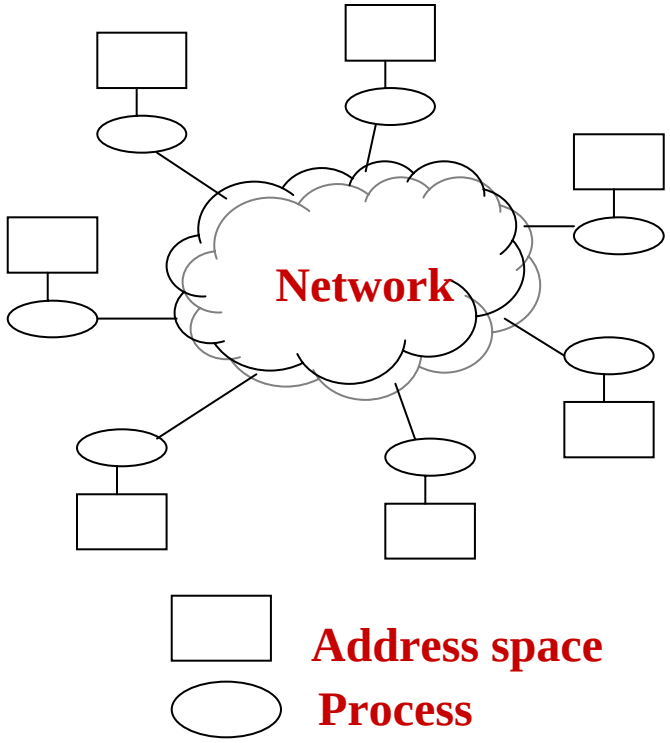
The Data Parallel Model



Different Data / address spaces

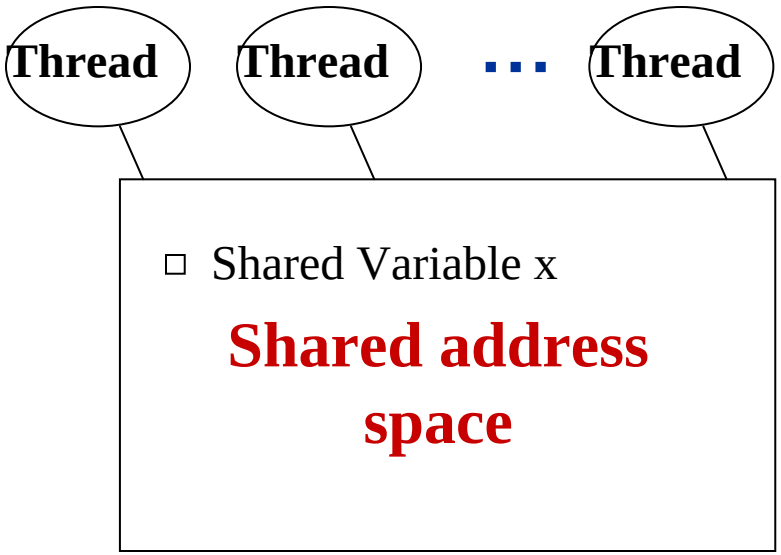
- Easy to write and comprehend, no synchronization required
- No independent branching
- Example: HPF

The Message Passing Model



- Programmers control data and work distribution
- Explicit communication, two-sided
- Library-based
- Excessive buffering
- Significant communication overhead for small transactions
- Example: MPI

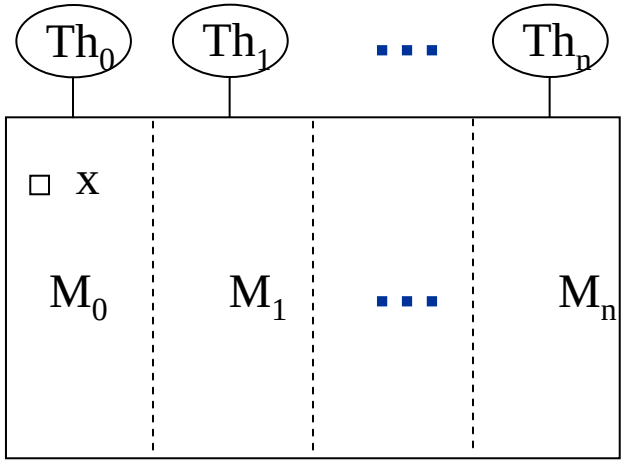
The Shared Memory Model



- 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

Partitioned Global Address Space



- 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! UPC, CAF, and Titanium

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...

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

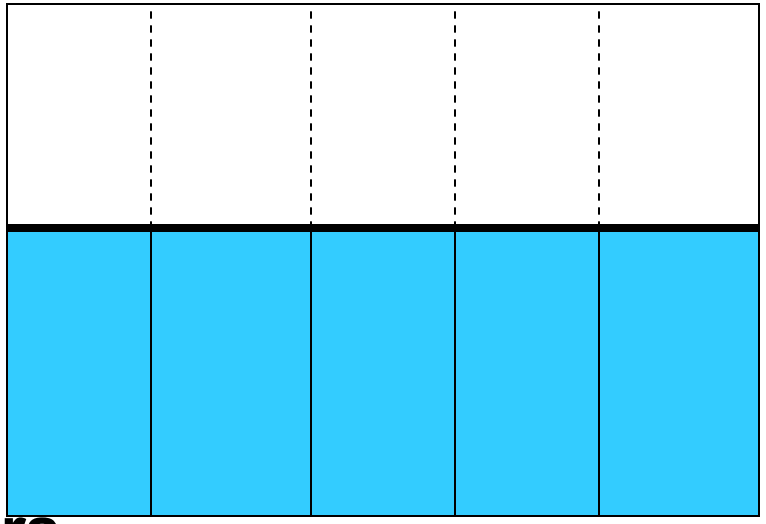
- BSP -- Bulk Synchronous Protocol
  - Alternating compute-communicate
- Global Arrays
  - Toolkit approach
  - Includes locality concepts

DSM/PGAS Model: Program View

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

DSM 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

DSM 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
  - Some collective library calls

DSM 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
- 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

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

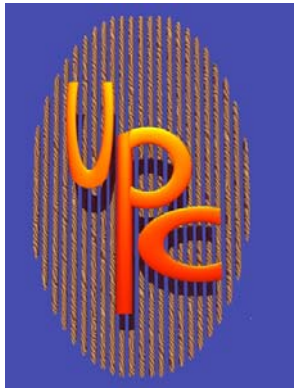
- Reference combining
- Compiler/runtime directed caching
- Remote memory operations

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



upc.gwu.edu

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, and may need to get close to hardware
    - to get performance, but
    - can get in trouble
  - Concise and efficient syntax
- Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C

Design Philosophy

- Start with C, 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

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, and DoD
- The participants currently are: ARSC, Compaq, CSC, Cray Inc., Etnus, GWU, HP, IBM, IDA CSC, Intrepid Technologies, LBNL, LLNL, MTU, NSA, UCB, UMCP, U florida, US DoD, US DoE

Status

- Specification v1.0 completed February of 2001, v1.1 in March 2003
- Benchmarking: Stream, GUPS, NPB suite, Splash-2, and others
- Testing suite v1.0, v1.1
- 2-Day Course offered in the US and abroad
- Research Exhibits at SC 2000-2002
- UPC web site: upc.gwu.edu
- UPC Book by SC 2004?

Hardware Platforms

- UPC implementations are available for
  - Cray T3D/E
  - Compaq AlphaServer SC
  - SGI O 2000/3000
  - Beowulf Reference Implementation
  - UPC Berkeley Compiler: Myrinet Clusters
  - Cray X-1
- Other ongoing and future implementations
  - UPC Berkeley Compiler: IBM SP and Quadrics, and Infiniband Clusters
  - HP Superdome
  - SGI and T3E 64-bit GCC

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

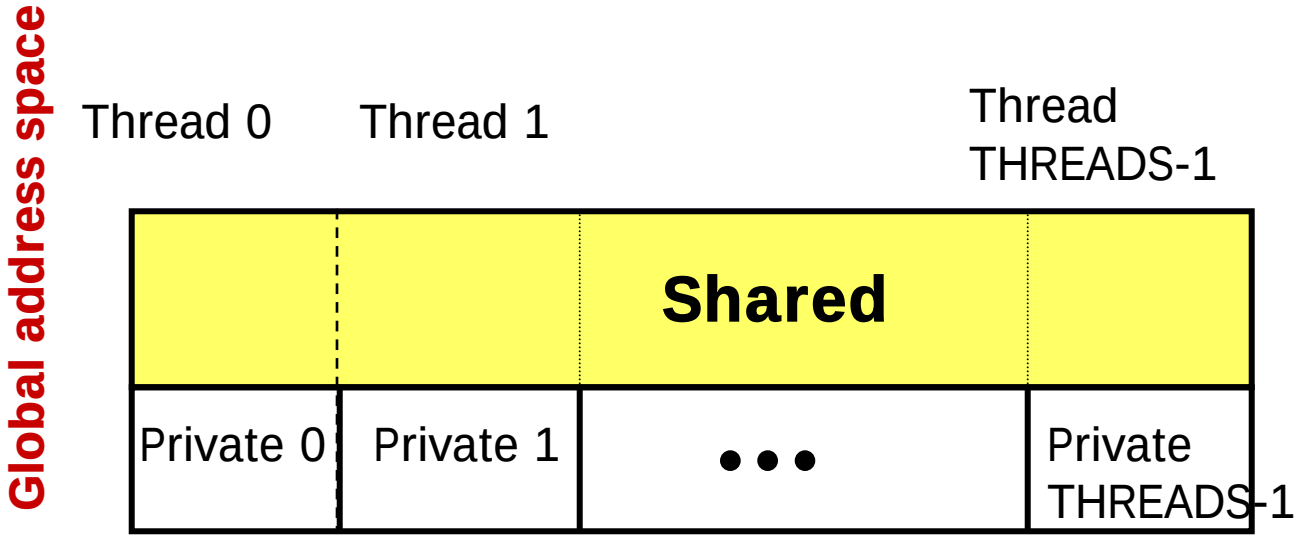
UPC Execution Model

- A number of threads working independently in a SPMD fashion
  - 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 pointer-to-shared 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 */
- Shared data cannot have dynamic scope

UPC Pointers

- **Pointer declaration:**

  shared int *p;

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

- **p is called a pointer to shared.**

A Third Example: Pointers to Shared

```
#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()

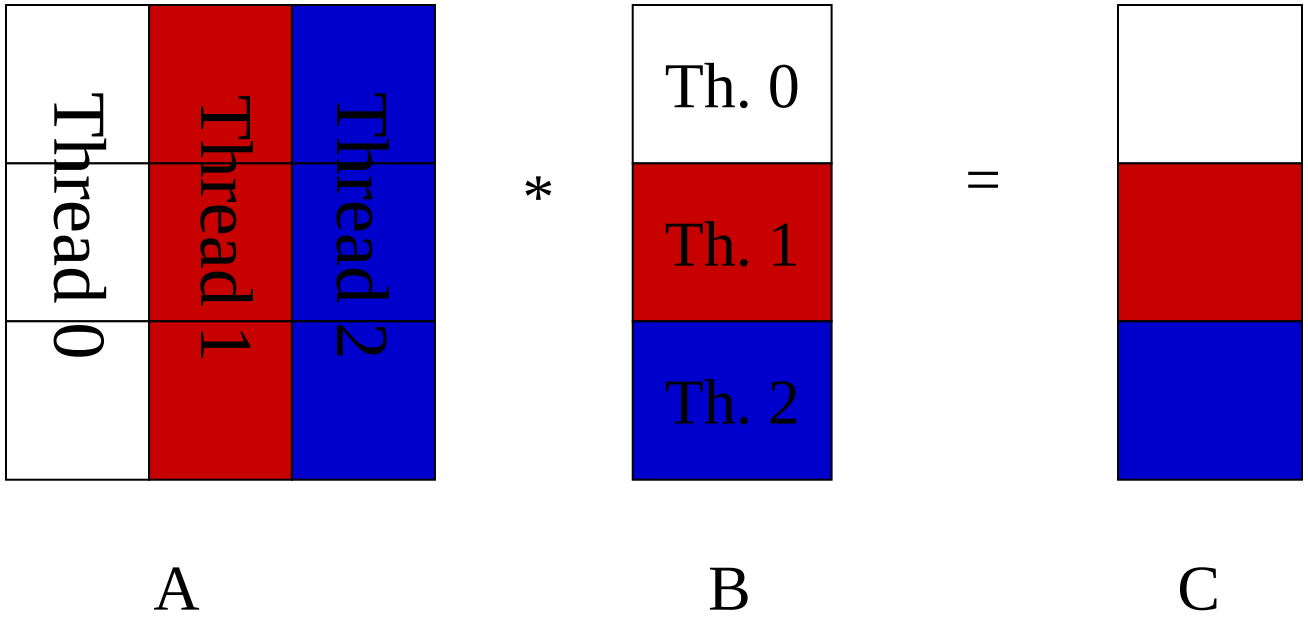
- Distributes independent iterations
- Each thread gets a bunch of iterations
- Affinity (expression) field to determine how to distribute work
- Simple C-like syntax and semantics

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

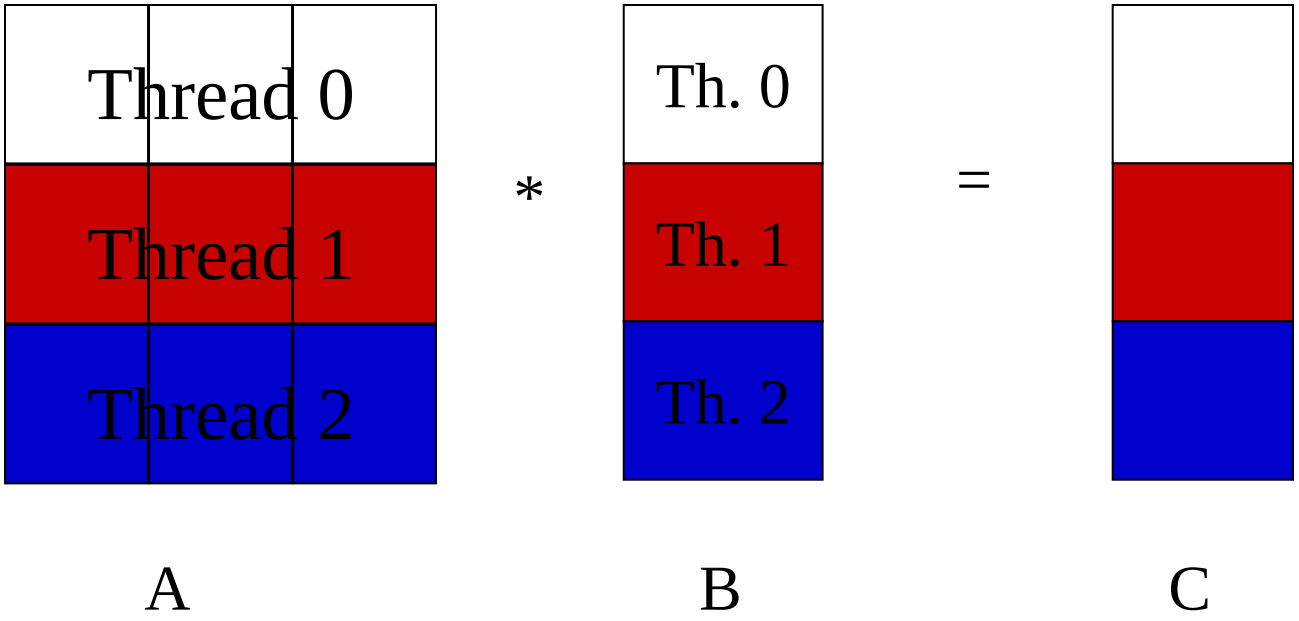
Example 4: UPC Matrix-Vector Multiplication- Default Distribution

```
// vect_mat_mult.c
#include <upc_relaxed.h>

shared int a[THREADS][THREADS] ;
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];
        }
}
```

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

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

Shared and Private Data

Examples of Shared and Private Data Layout:

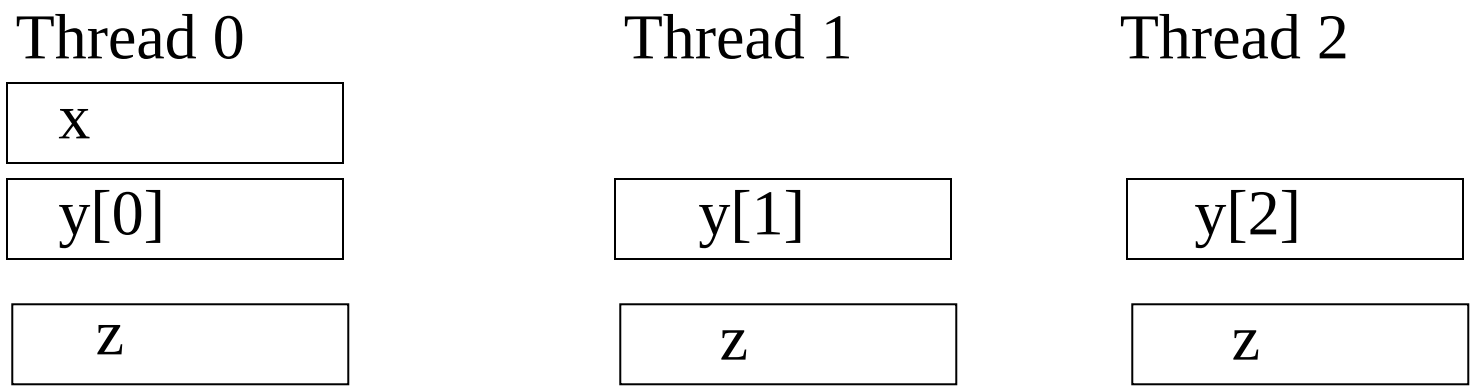
Assume THREADS = 3

shared int x; /*x will have affinity to thread 0 */

shared int y[THREADS];

int z;

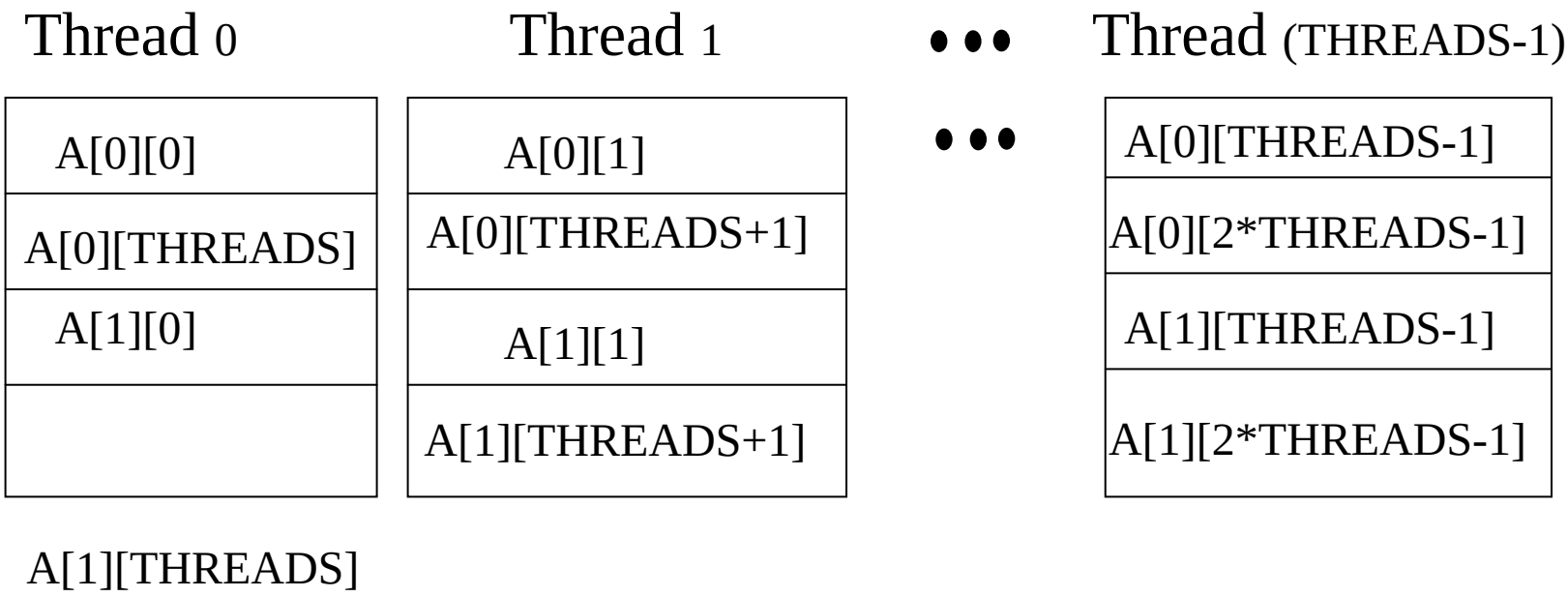
will result in the layout:



Shared and Private Data

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

will result in the following data layout:



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:**

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

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]			

UPC Pointers

Where does the pointer reside?

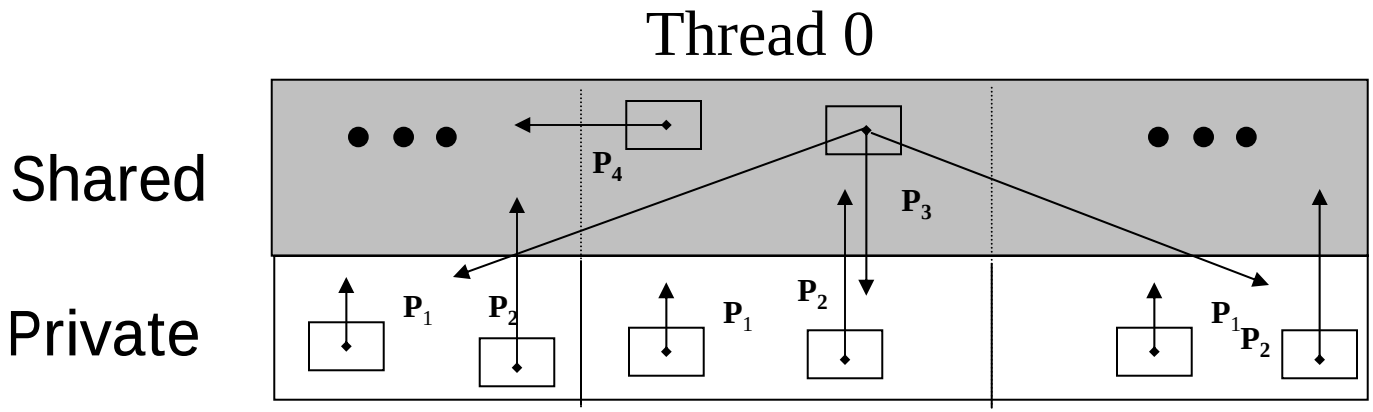
Where does it point?

	Private	Shared
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 */
- **You may find many using "shared pointer" to mean a pointer pointing to a shared object, e.g. equivalent to p2 but could be p4 as well.**

UPC Pointers

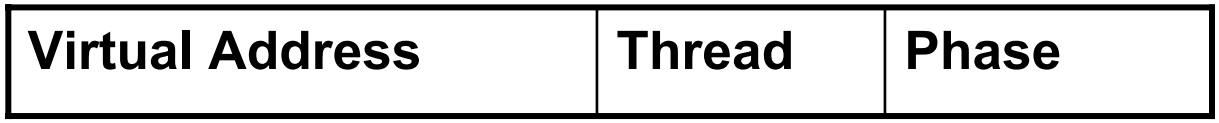


UPC Pointers

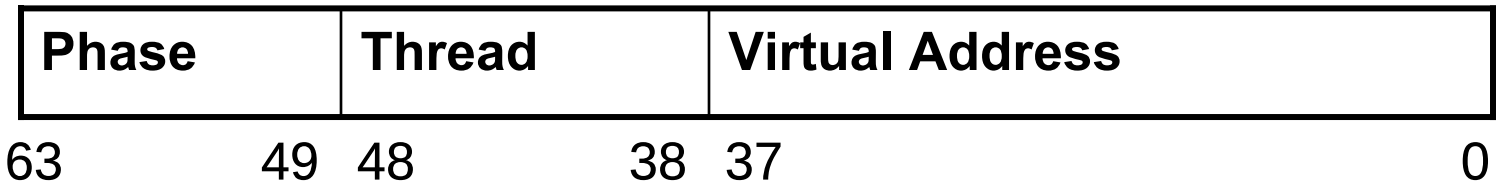
- What are the common usages?
  - int *p1; /* access to private data or to local shared data */
  - shared int *p2; /* independent access of threads to data in shared space */
  - int *shared p3; /* not recommended*/
  - shared int *shared p4; /* common access of all threads to data in the shared space*/

UPC Pointers

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



- Example: Cray T3E implementation



UPC Pointers

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

Special Functions

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

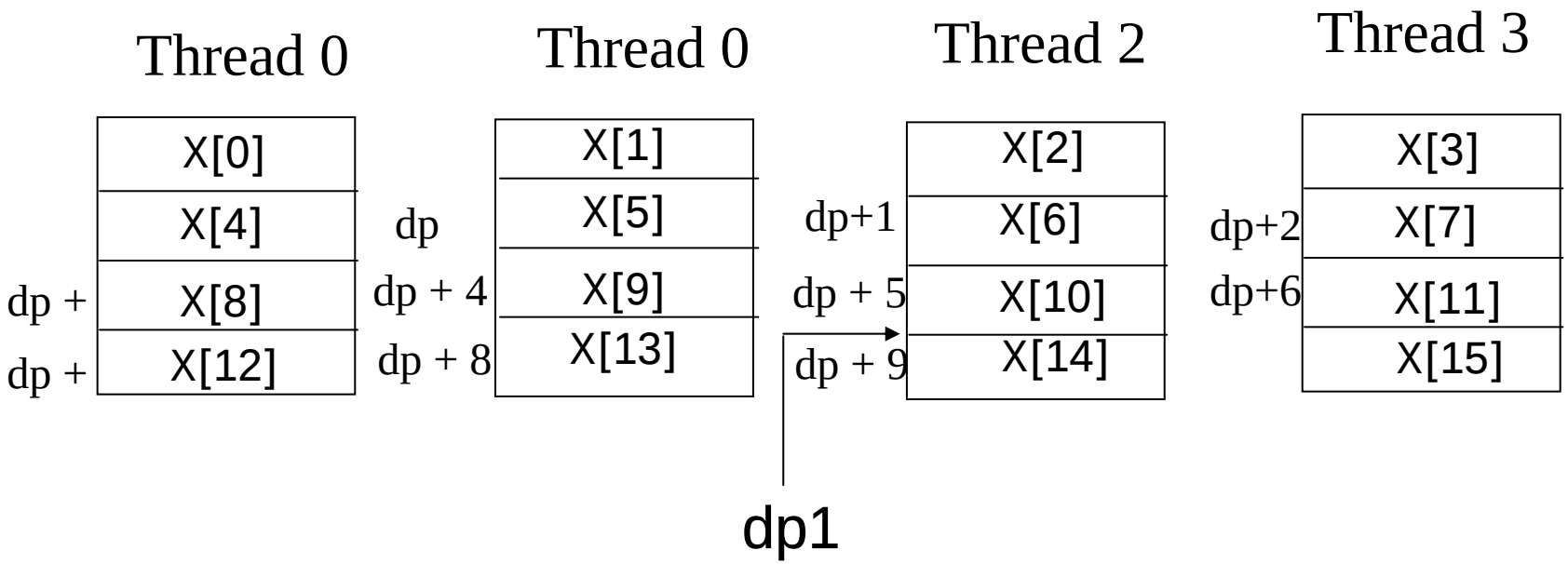
UPC Pointers

pointer to shared Arithmetic Examples:

Assume THREADS = 4

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

UPC Pointers

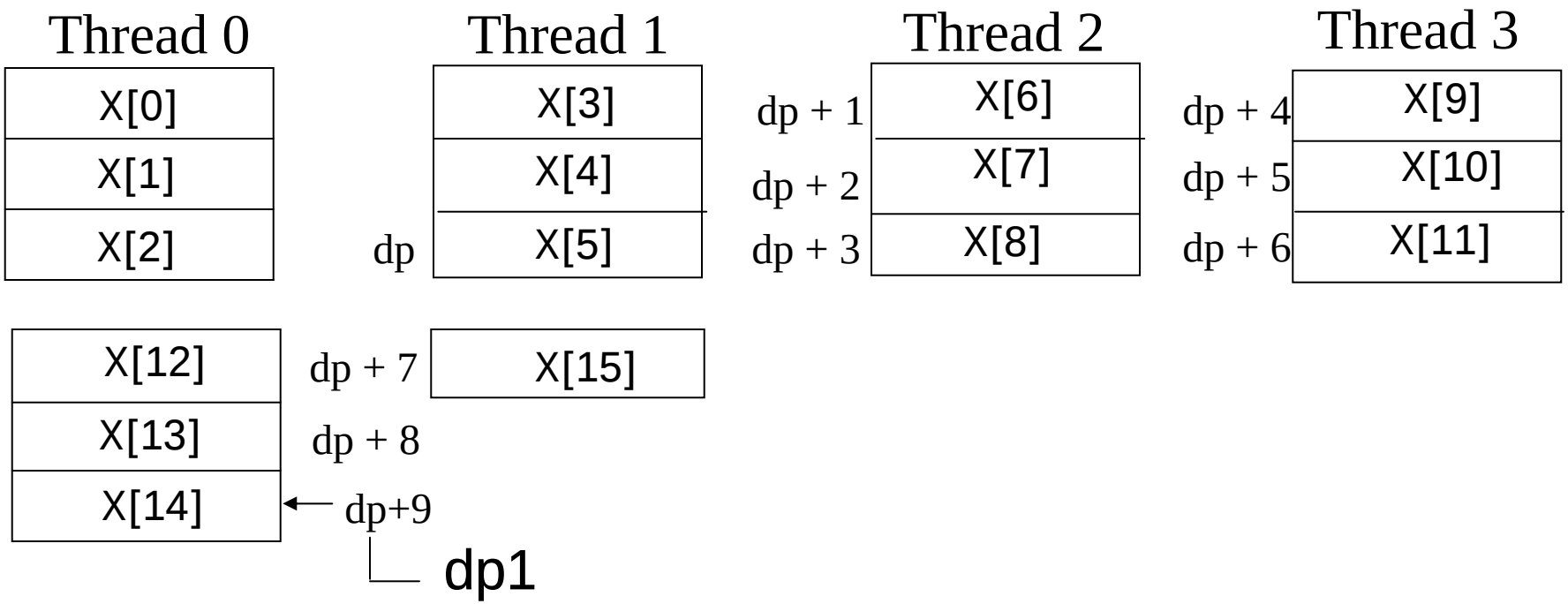


UPC Pointers

Assume THREADS = 4

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

UPC Pointers



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

Work Sharing: upc_forall()

- **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 and Pointers
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]

Memory Freeing

```
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**

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

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

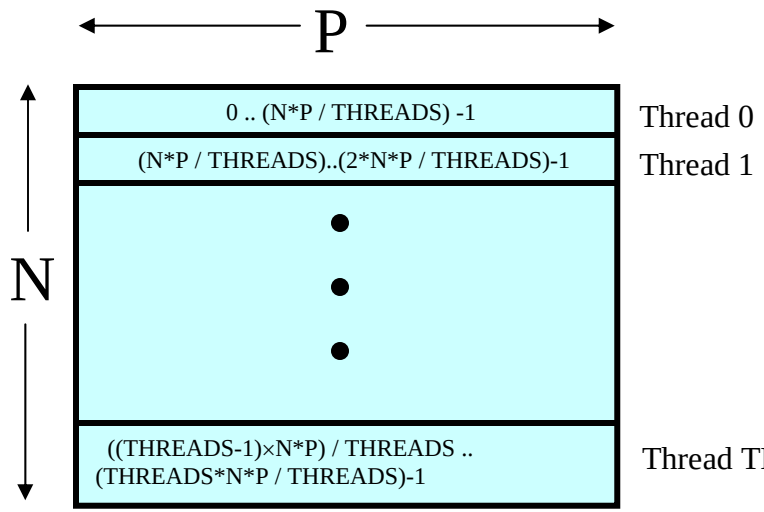
```
#include <stdlib.h>
#include <time.h>
#define N  4
#define P  4
#define M 4
int a[N][P] =  {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

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];
            }
      }
}
```

Note: most compilers are not yet supporting the intialization in declaration statements

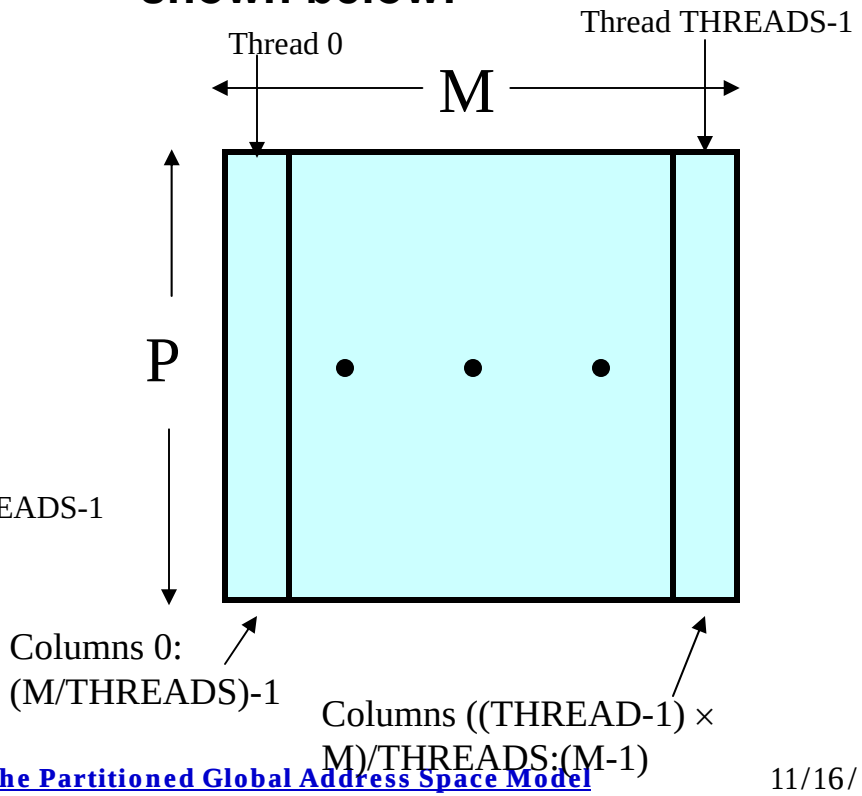
Domain Decomposition for UPC

- 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

```
#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

```
#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];
                }
        }
}
```

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

Synchronization

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

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- Fence

- Upc provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    - upc_fence;
  - UPC 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

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 and Pointers
6. Dynamic Memory Management
7. Programming Examples
8. Synchronization
9. Performance Tuning and Early Results
10. Concluding Remarks

Productivity ~ Code Size

		SEQ[*1]	MPI	SEQ[*2]	UPC	MPI/SEQ (%)	UPC/SEQ (%)
GUPS	#line	41	98	41	47	139.02	14.63
	#char	1063	2979	1063	1251	180.02	17.68
Histogram	#line	12	30	12	20	150.00	66.67
	#char	188	705	188	376	275.00	100.00
NAS-EP	#line	130	187	127	149	43.85	17.32
	#char	4741	6824	2868	3326	44.94	15.97
NAS-FT	#line	704	1281	607	952	81.96	56.84
	#char	23662	44203	13775	20505	86.81	48.86
N-Queens	#line	86	166	86	139	93.02	61.63
	#char	1555	3332	1555	2516	124.28	61.80

All the line counts are the number of real code lines (no comments, no blocks)

[*1]: The sequential code is coded in C except for NAS-EP and FT which are coded in Fortran.
[*2]: The sequential code is always in C.

How to Exploit the Opportunities for Performance Enhancement?

- Compiler optimizations
- Run-time system
- Hand tuning

List of Possible Optimizations for UPC Codes

- Space privatization: use private pointers 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: For example, overlap remote accesses with local processing using split-phase barriers
- Vendors can also help decrease cost for address translation and providing optimized standard libraries

Performance of Shared vs. Private Accesses (Old COMPAQ Measurement)

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

Recent compiler developments have improved some of that

Using Local Pointers Instead of pointer to shared

```
…
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 pointer to shared
- The pointer dereference can be one order of magnitude faster

Performance of UPC

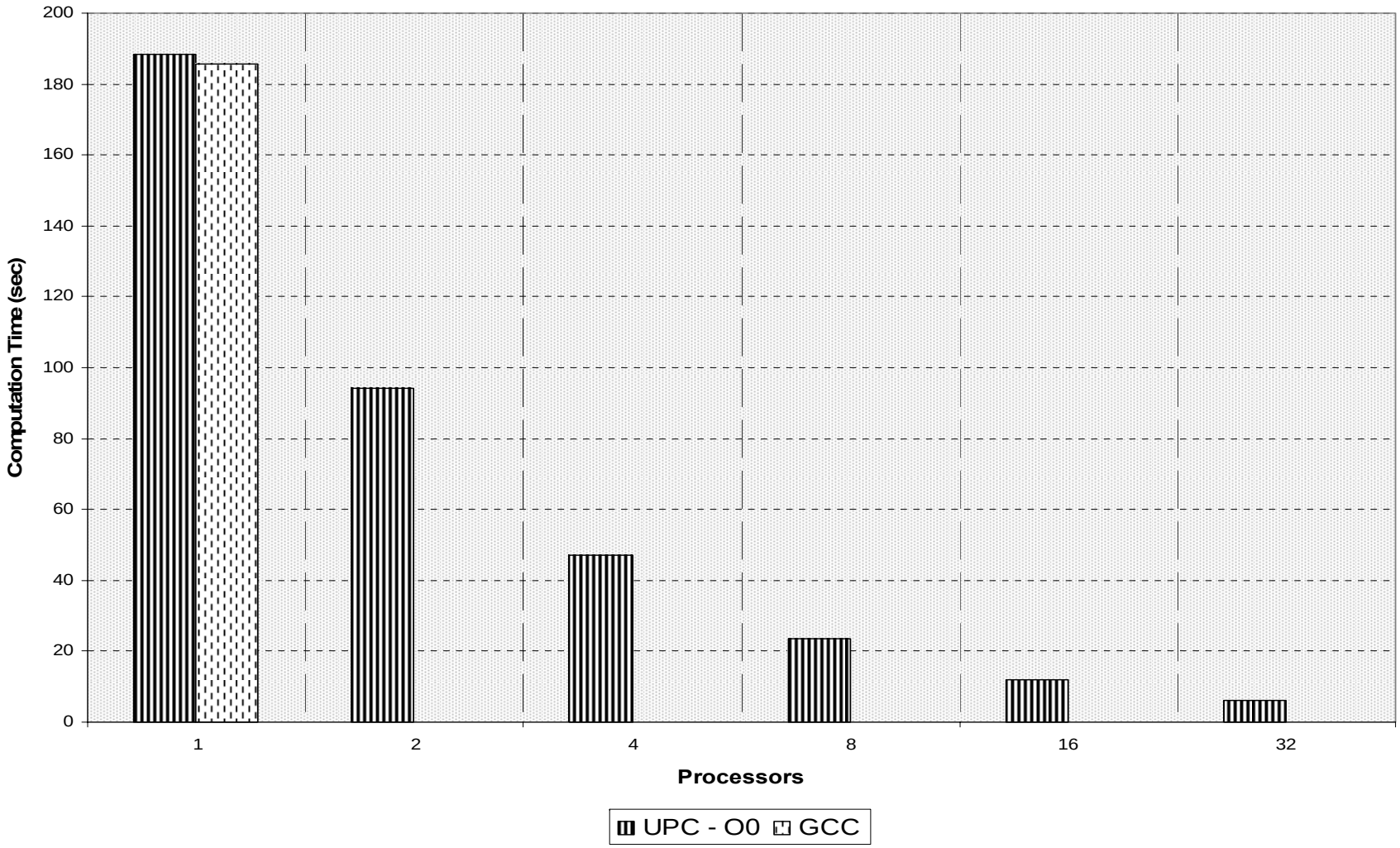
- UPC benchmarking results
  - Nqueens Problem
  - Matrix Multiplication
  - Sobel Edge detection
  - Stream and GUPS
  - NPB
  - Splash-2
- Compaq AlphaServer SC and Origin 2000/3000
- Check the web site for new measurements

Shared vs. Private Accesses (Recent SGI Origin 3000 Measurement)

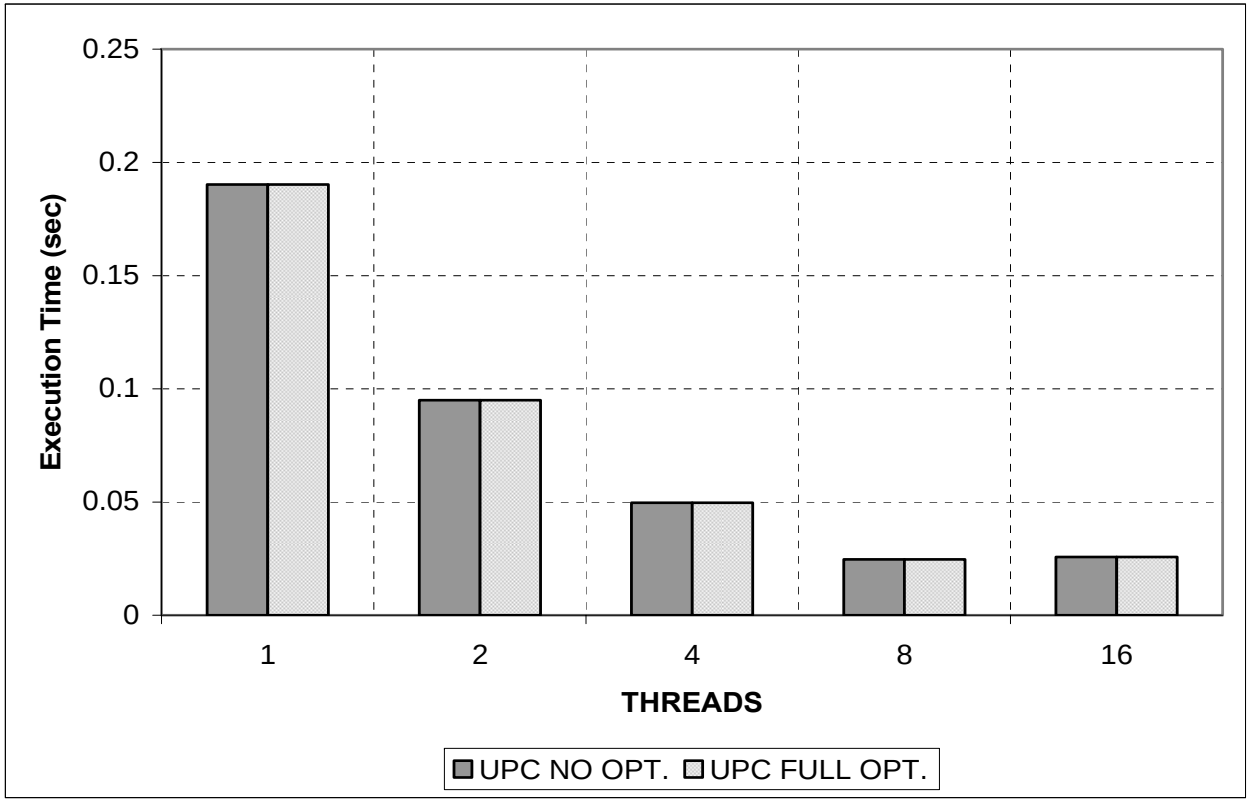
STREAM BENCHMARK

MB/S	Memcpy	Array Copy	Scale	Sum	Block Get	Block Scale
GCC	400	266	266	800	N/A	N/A
UPC Private	400	266	266	800	N/A	N/A
UPC Local	N/A	40	44	100	400	400
UPC Shared (SMP)	N/A	40	44	88	266	266
UPC Shared (Remote)	N/A	34	38	72	200	200

Execution Time over SGI–Origin 2k NAS-EP – Class A

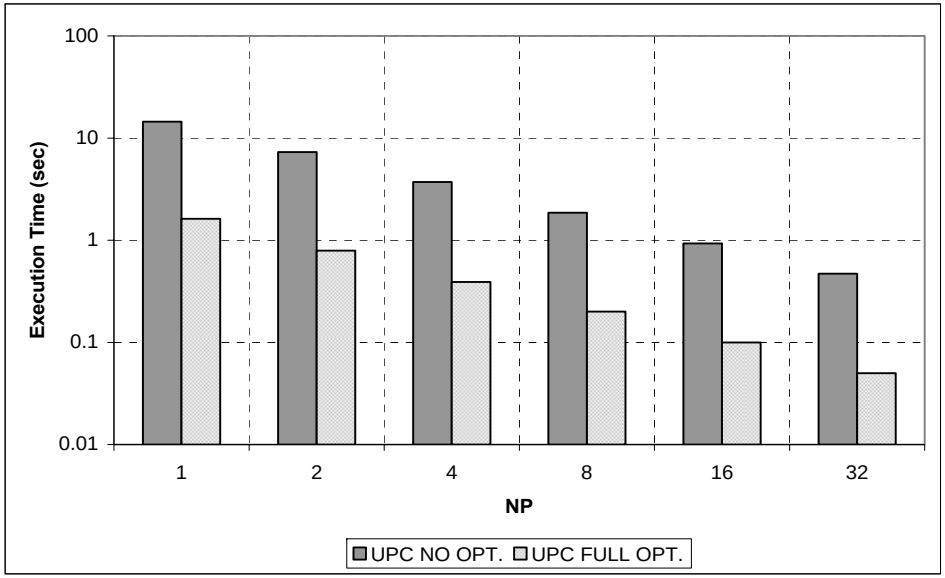


Performance of the N-QUEENS problem on the Origin 2000

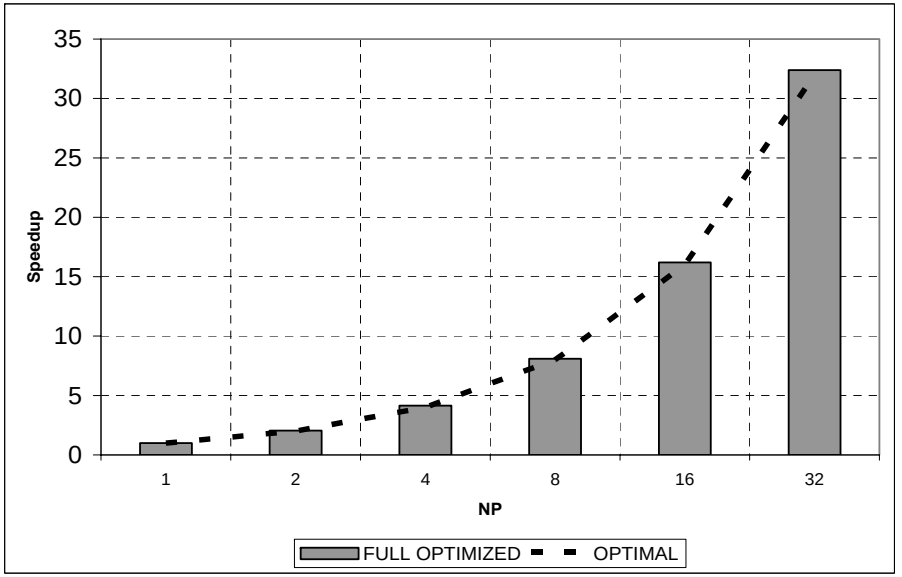


UPC N-Queens:
Execution Time

Performance of Edge detection on the Origin 2000

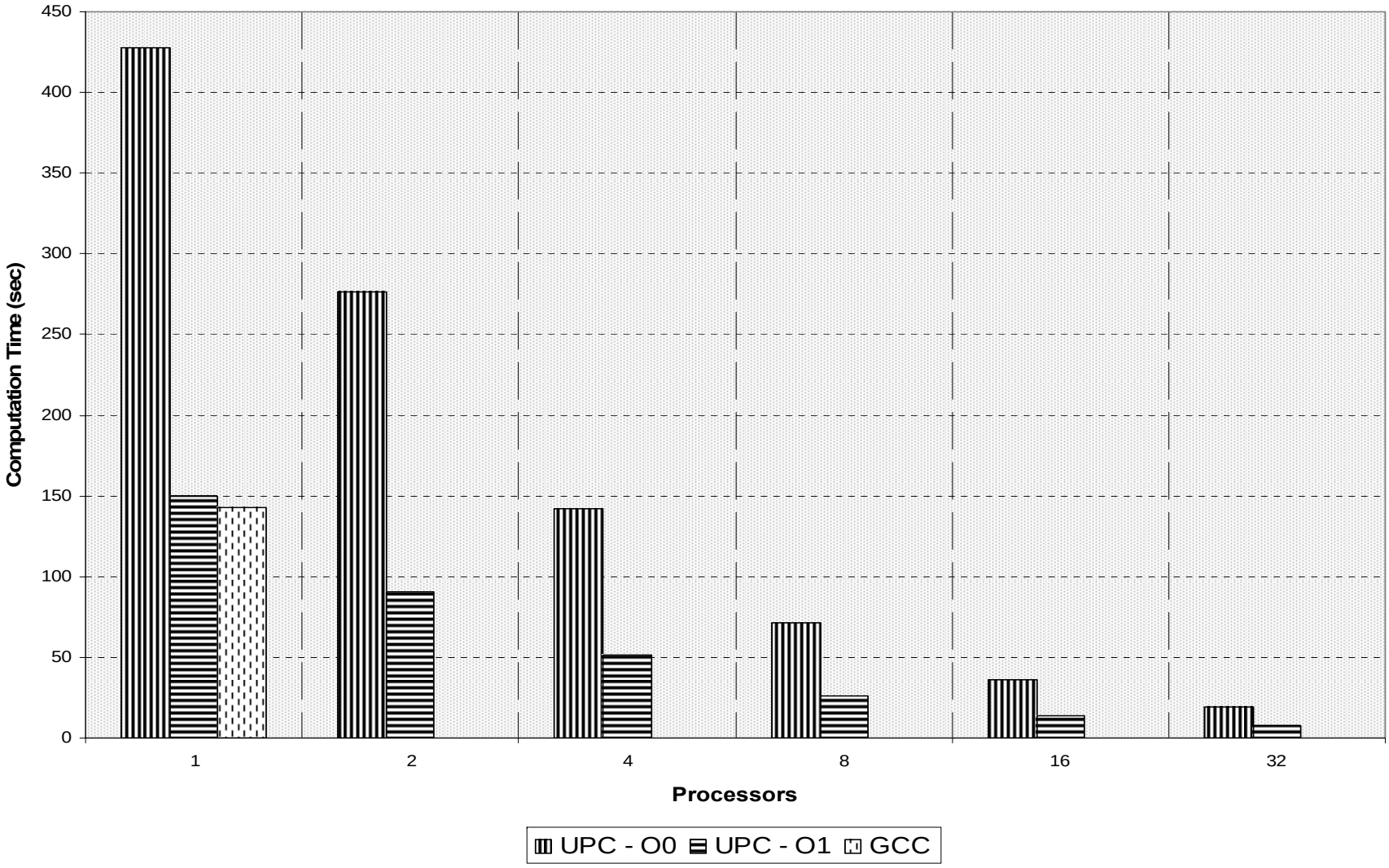


Execution Time

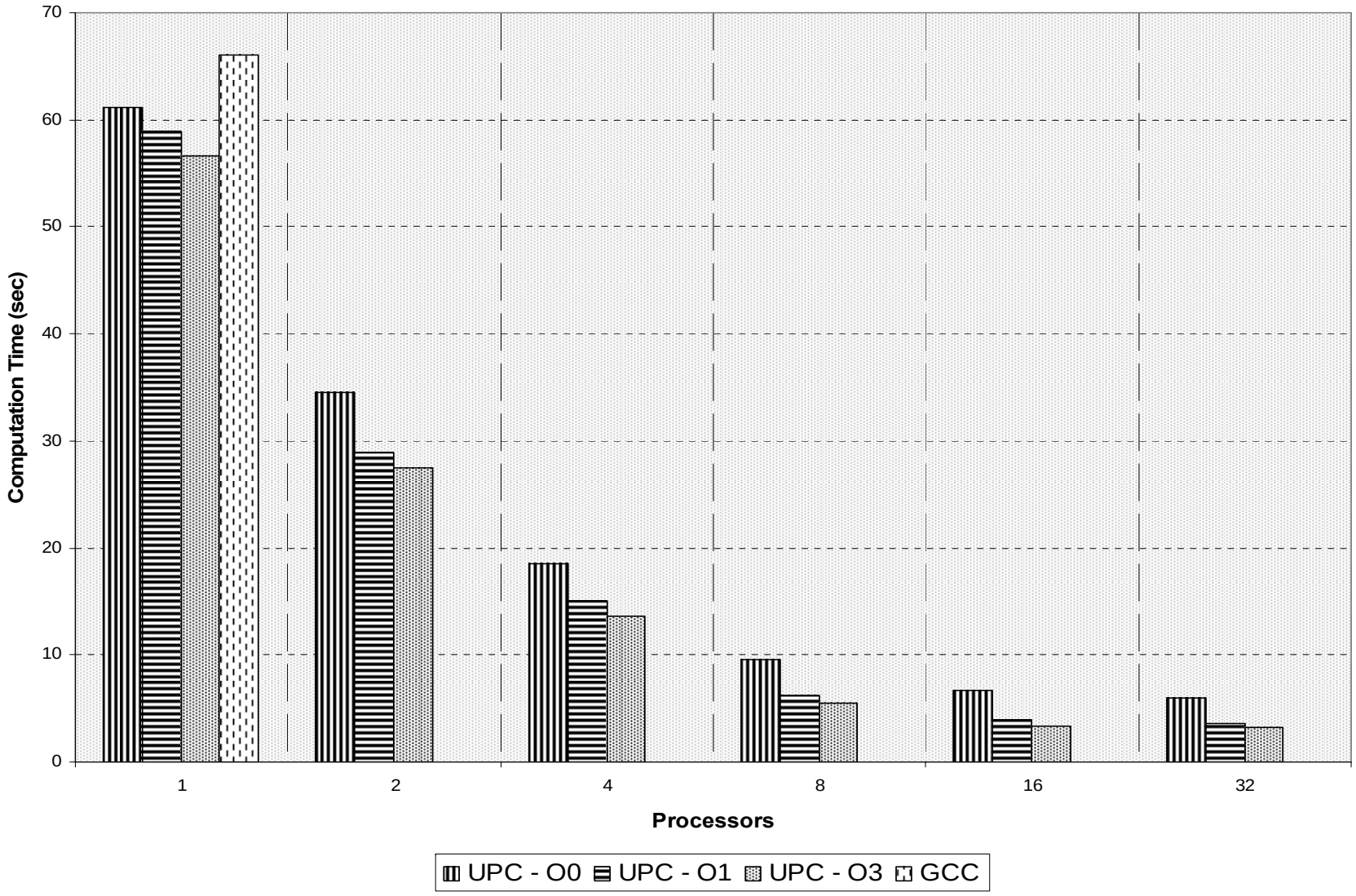


Speedup

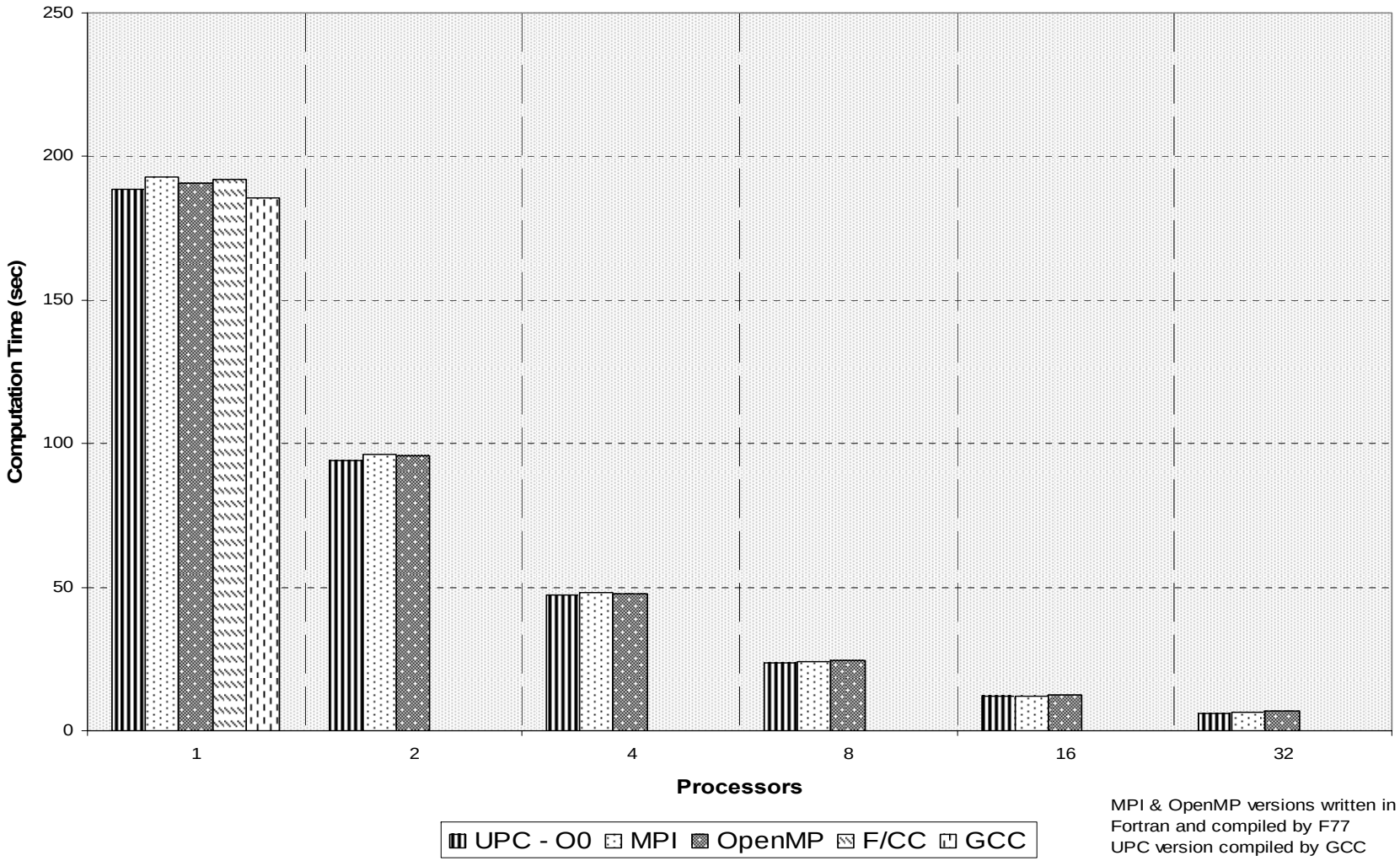
Execution Time over SGI–Origin 2k NAS-FT – Class A



Execution Time over SGI–Origin 2k NAS-CG – Class A

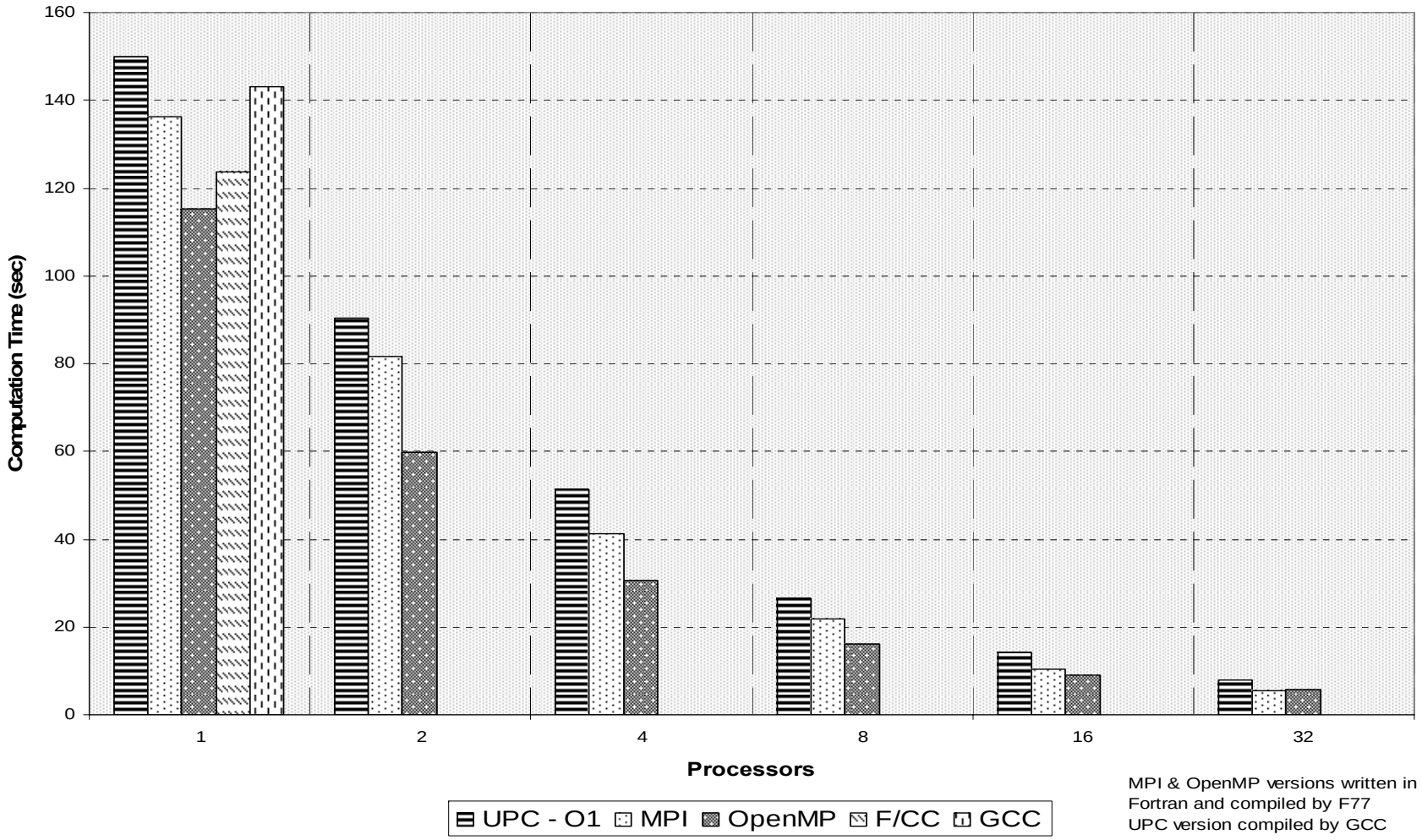


Execution Time over SGI–Origin 2k NAS-EP – Class A

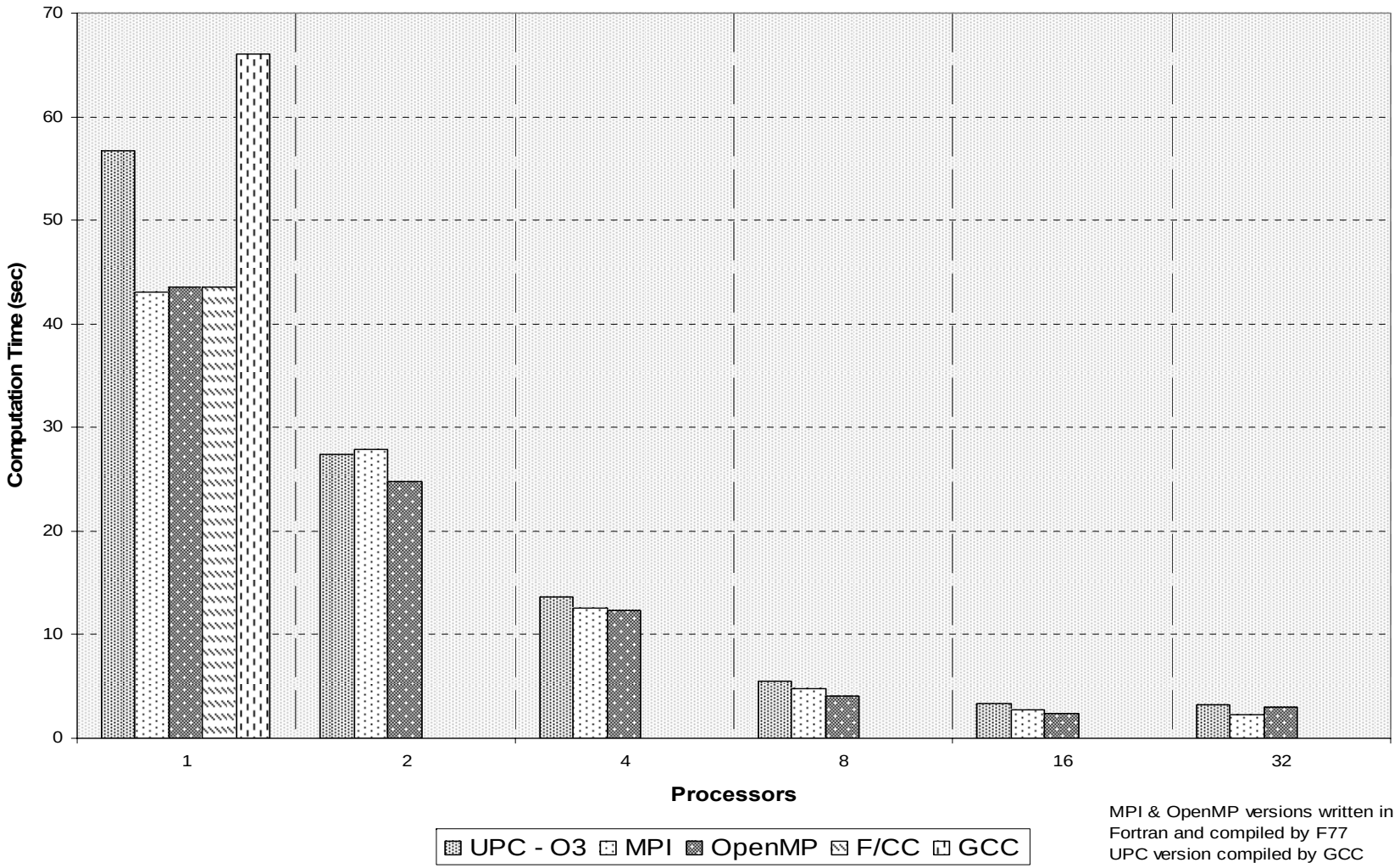


MPI & OpenMP versions written in Fortran and compiled by F77
UPC version compiled by GCC

Execution Time over SGI–Origin 2k NAS-FT – Class A

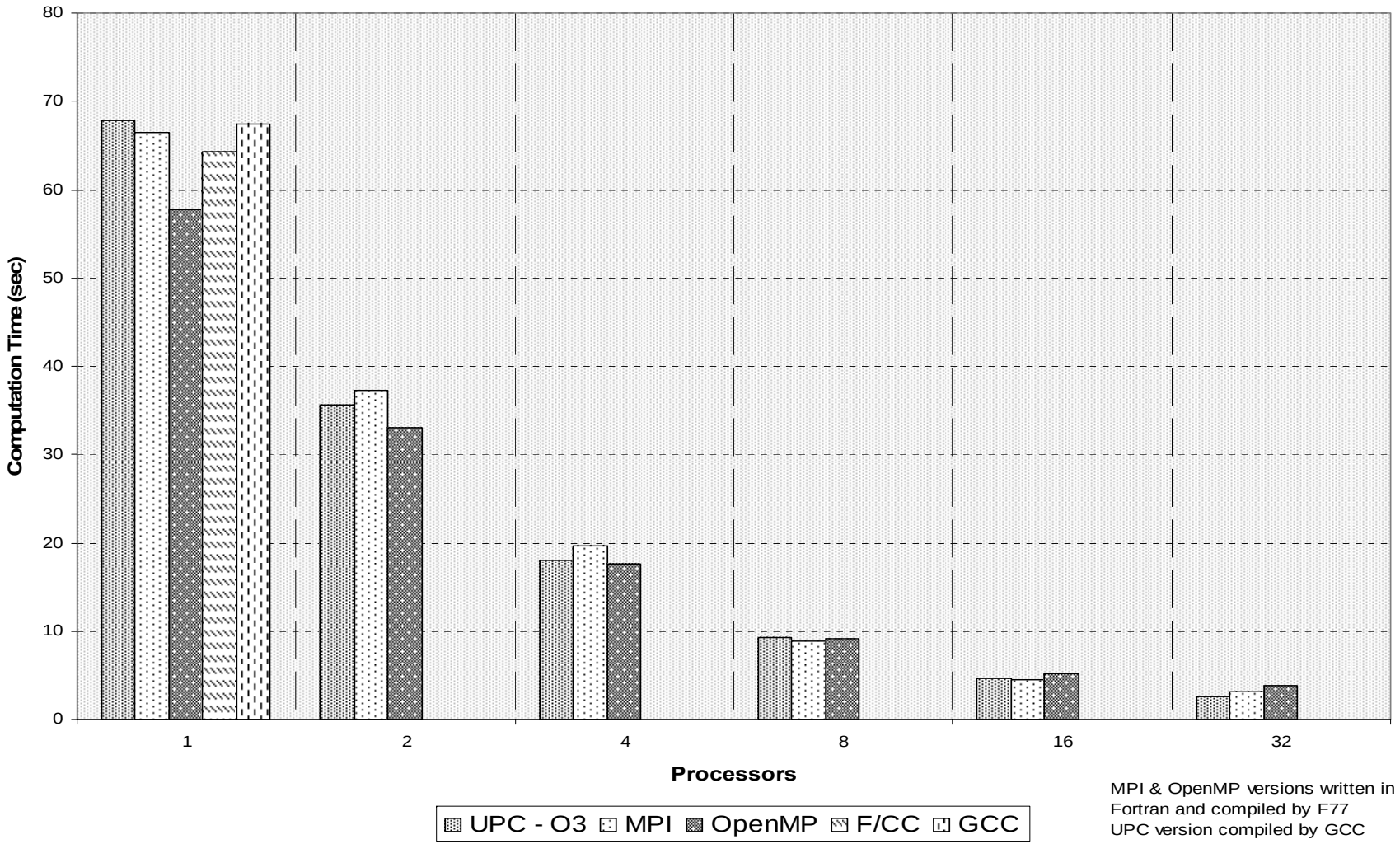


Execution Time over SGI–Origin 2k NAS-CG – Class A



MPI & OpenMP versions written in Fortran and compiled by F77
UPC version compiled by GCC

Execution Time over SGI–Origin 2k NAS-MG – Class A



MPI & OpenMP versions written in Fortran and compiled by F77
UPC version compiled by GCC

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

Conclusions

$UPC_{Time\text{-}To\text{-}Solution} =$

$UPC_{Programming\ Time} + UPC_{Execution\ Time}$

- Simple and Familiar View
  - Domain decomposition maintains global application view
  - No function calls
- Concise Syntax
  - Remote writes with assignment to shared
  - Remote reads with expressions involving shared
  - Domain decomposition (mainly) implied in declarations (logical place!)

- Data locality exploitation
- No calls
- One-sided communications
- Low overhead for short accesses

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
- Hand tuned code, with block moves, is still substantially simpler than message passing code

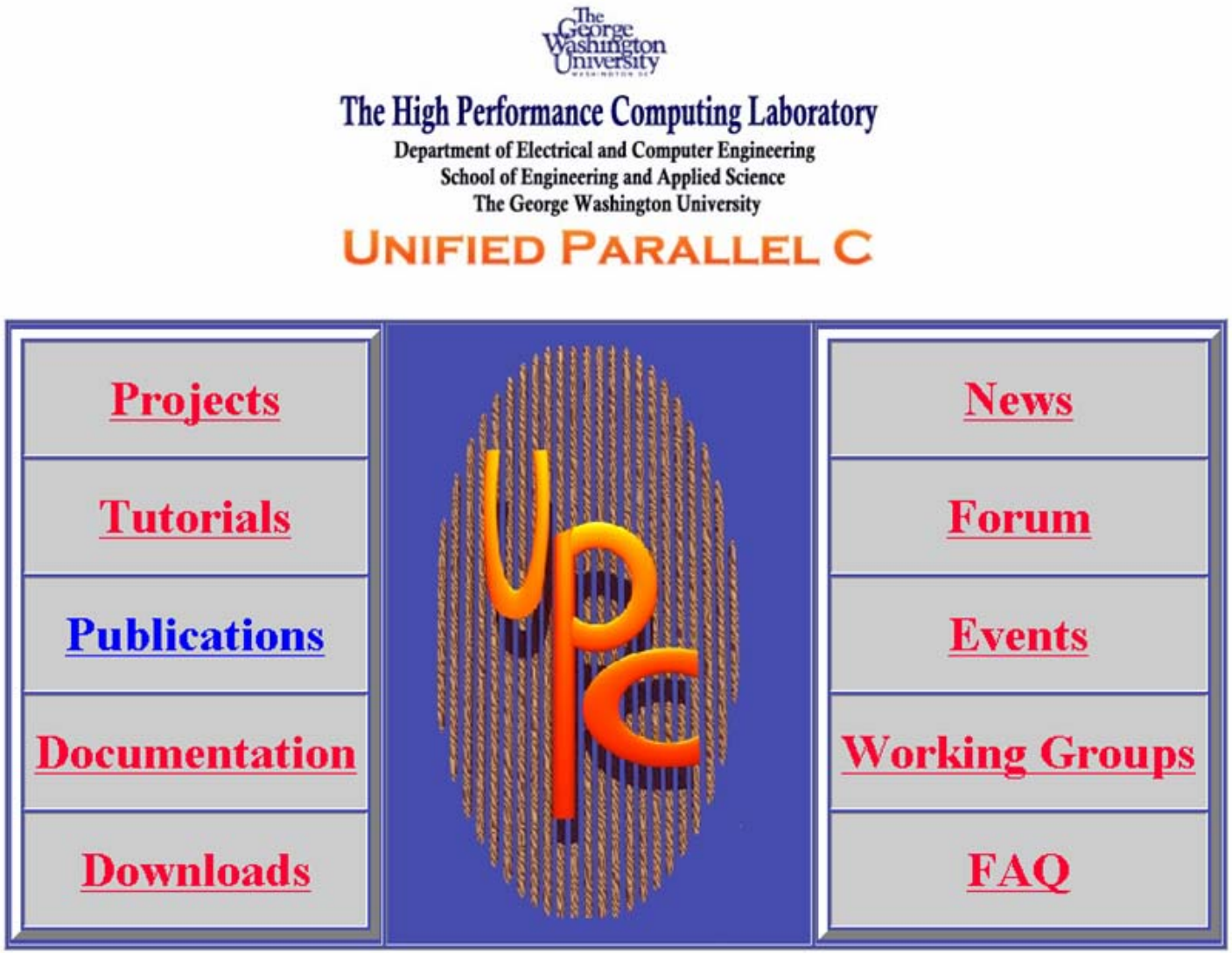
Conclusions

- Automatic compiler optimizations should focus on
  - Inexpensive address translation
  - Space Privatization for local shared accesses
  - Prefetching and aggregation of remote accesses, prediction is easier under the UPC model
- More performance help is expected from optimized standard library implementations, specially collective and I/O

References

- The official UPC website, http://upc.gwu.edu
- T. A.El-Ghazawi, W.W.Carlson, J. M. Draper. UPC Language Specifications V1.1 (http://upc.gwu.edu). May, 2003
- François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed S. Mohamed, Tarek A. El-Ghazawi Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture, International Parallel and Distributed Processing Symposium(IPDPS'03) Nice Acropolis Convention Center, Nice, France, 2003.
- Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, Katherine Yelick, A performance analysis of the Berkeley UPC compiler, International Conference on Supercomputing, Proceedings of the 17th annual international conference on Supercomputing 2003,San Francisco, CA, USA
- Tarek A. El-Ghazawi, François Cantonnet, UPC Performance and Potential: A NPB Experimental Study, SuperComputing 2002 (SC2002). IEEE, Baltimore MD, USA, 2002.
- Tarek A.El-Ghazawi, Sébastien Chauvin, UPC Benchmarking Issues, Proceedings of the International Conference on Parallel Processing (ICPP'01). IEEE CS Press. Valencia, Spain, September 2001.

http://upc.gwu.edu



Co-Array Fortran Tutorial SC 2003

Robert W. Numrich

Minnesota Supercomputing Institute

University of Minnesota

rwn@msi.umn.edu

Abstract

Co-Array Fortran is a simple extension to Fortran 90 that allows programmers to write efficient parallel applications using a Fortran-like syntax. It assumes the SPMD programming model with replicated data objects called co-arrays. Co-Array objects are visible to all processors and each processor can read or write data belonging to any other processor by setting the index of the co-dimension to the appropriate value. It can be thought of as the SHMEM model implemented as an extension to the language. The combination of co-array syntax with the 'object-oriented' features of Fortran 90 provides a powerful method of encapsulating parallel data structures and parallel algorithms into Fortran 90 modules that resemble class libraries in an object-oriented language.

Outline

1. Philosophy of Co-Array Fortran
2. Execution model
3. Co-arrays and co-dimensions
4. Memory model
5. Relative image indices
6. CAF intrinsic procedures
7. Dynamic memory management
8. CAF I/O
9. "Object-Oriented" Techniques
10. Summary
11. Examples
    - Examples from Linear Algebra
    - Example from UK Met Office
12. Exercises
    - Global reductions
    - PIC code fragment
    - CAF Class Library
    - Poisson Solver

1. Philosophy of Co-Array Fortran

The Guiding Principle behind Co-Array Fortran

- 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?
- How can it be expressed so that existing compiler technology can implement it easily and efficiently?

What's the Problem with SPMD?

- One processor knows nothing about another's memory layout.
  - Local variables live on the local heap.
  - Addresses, sizes and shapes are different on different program images.
- 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 variables using an awkward mechanism**
- 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 Extension

- Incorporate the SPMD Model into Fortran 95
- Multiple images of the same program
  - Text and data are replicated in each image
- Mark some variables with co-dimensions
  - Co-dimensions behave like normal dimensions
  - Co-dimensions express a logical problem decomposition
  - One-sided data exchange between co-arrays using a Fortran-like syntax
- Require the underlying run-time system to map the logical problem decomposition onto specific hardware.

2. Execution Model

The CAF Execution Model

- The number of images is fixed and each image has its own index, retrievable at run-time:

  $1 \leq$ num_images()

  $1 \leq$ this_image() $\leq$ num_images()

- Each image executes the same program independently of the others.
- The programmer inserts explicit synchronization and branching as needed.
- An "object" has the same name in each image.
- Each image works on its own local data.
- An image moves remote data to local data through, and only through, explicit CAF syntax.

3. Co-Arrays and Co-Dimensions

What is Co-Array Syntax?

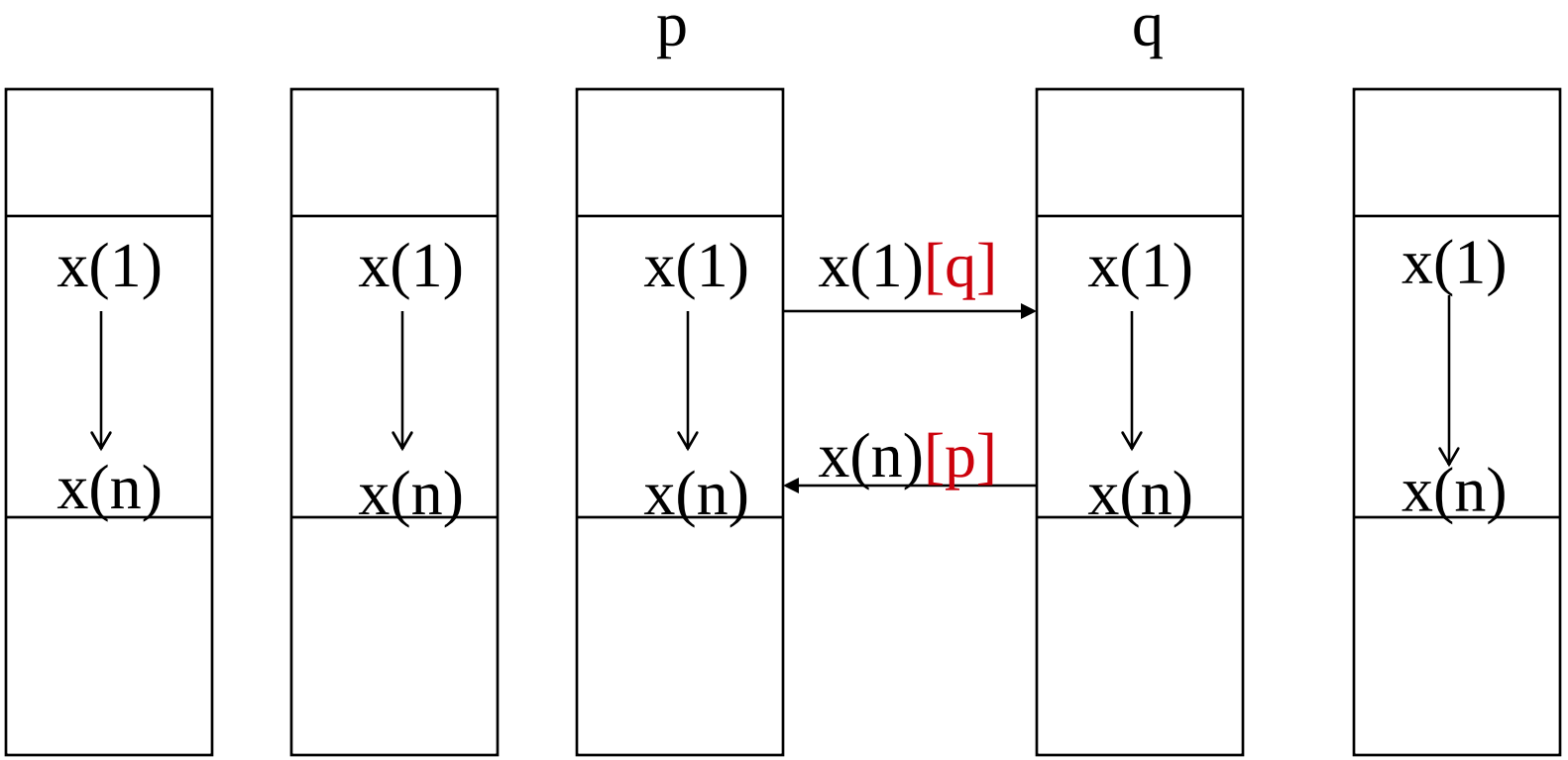
- Co-Array syntax is a simple parallel extension to normal Fortran syntax.
  - It uses normal rounded brackets ( ) to point to data in local memory.
  - It uses square brackets [ ] to point to data in remote memory.
  - Syntactic and semantic rules apply separately but equally to ( ) and [ ].

Examples of Co-Array Declarations

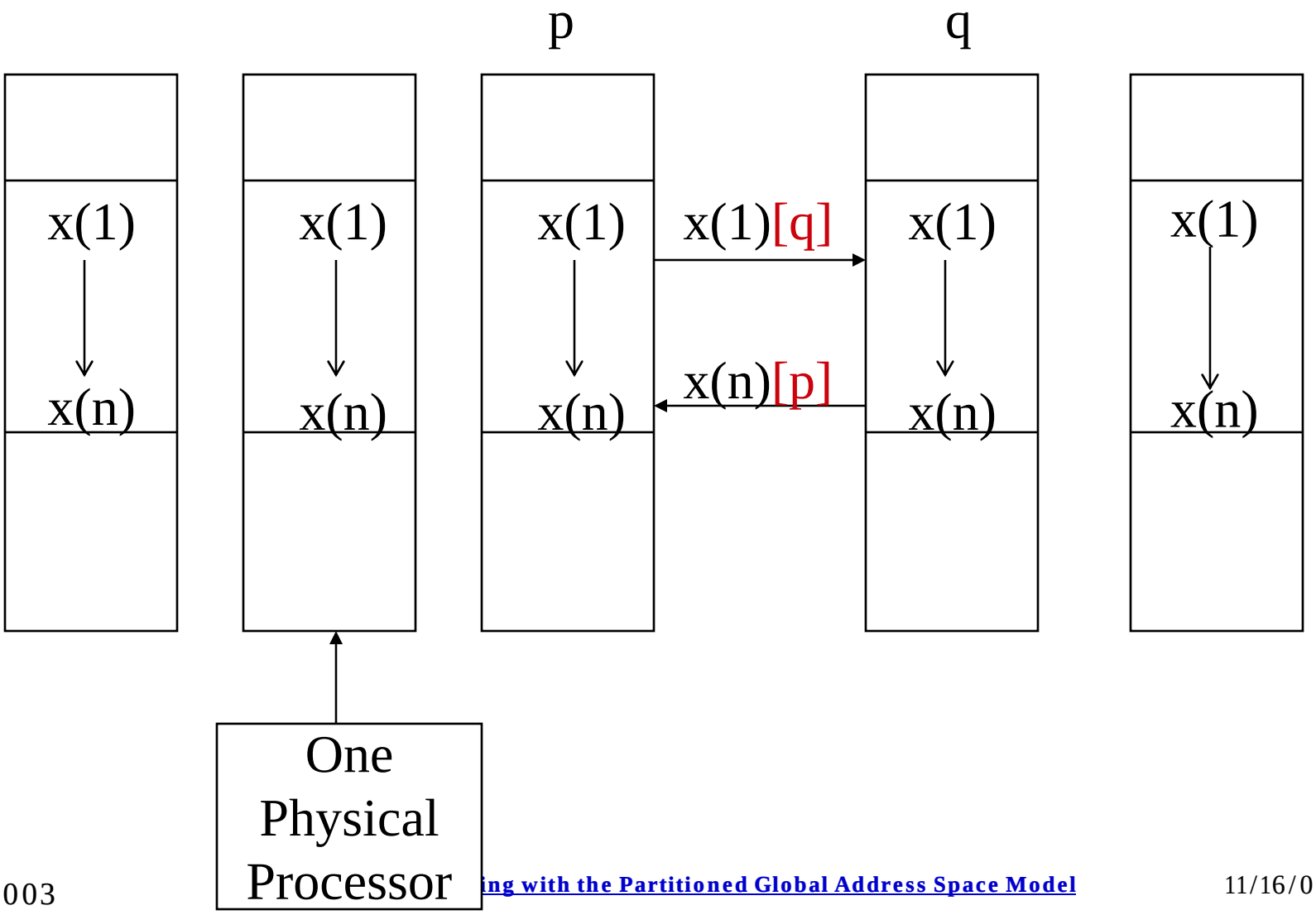
```
real :: a(n)[*]
complex :: z[*]
integer :: index(n)[*]
real :: b(n)[p, *]
real :: c(n,m)[0:p, -7:q, +11:*]
real,allocatable :: w(:)[:]
type(field) :: maxwell[p,*]
```

4. CAF Memory Model

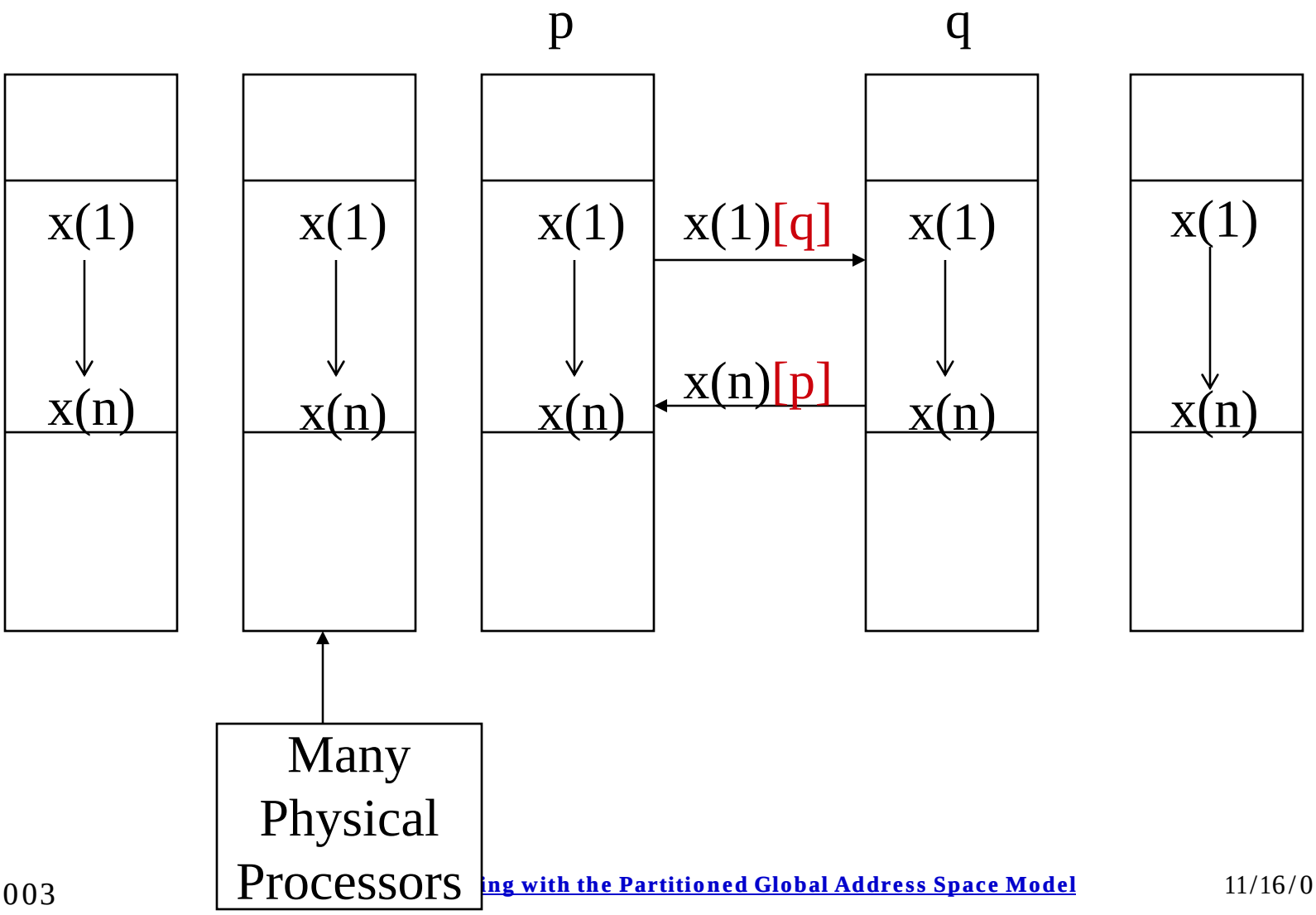
CAF Memory Model



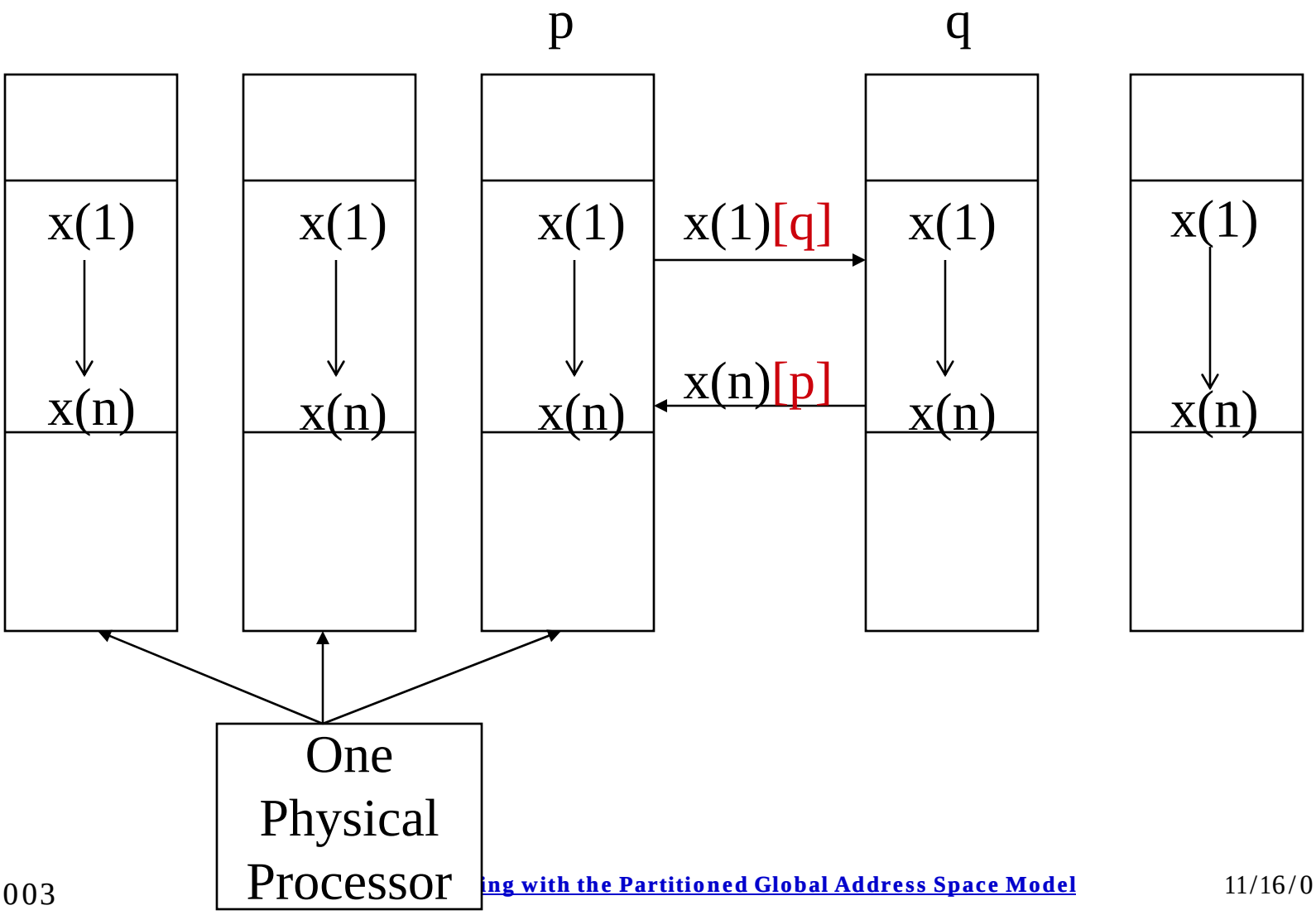
One-to-One Execution Model



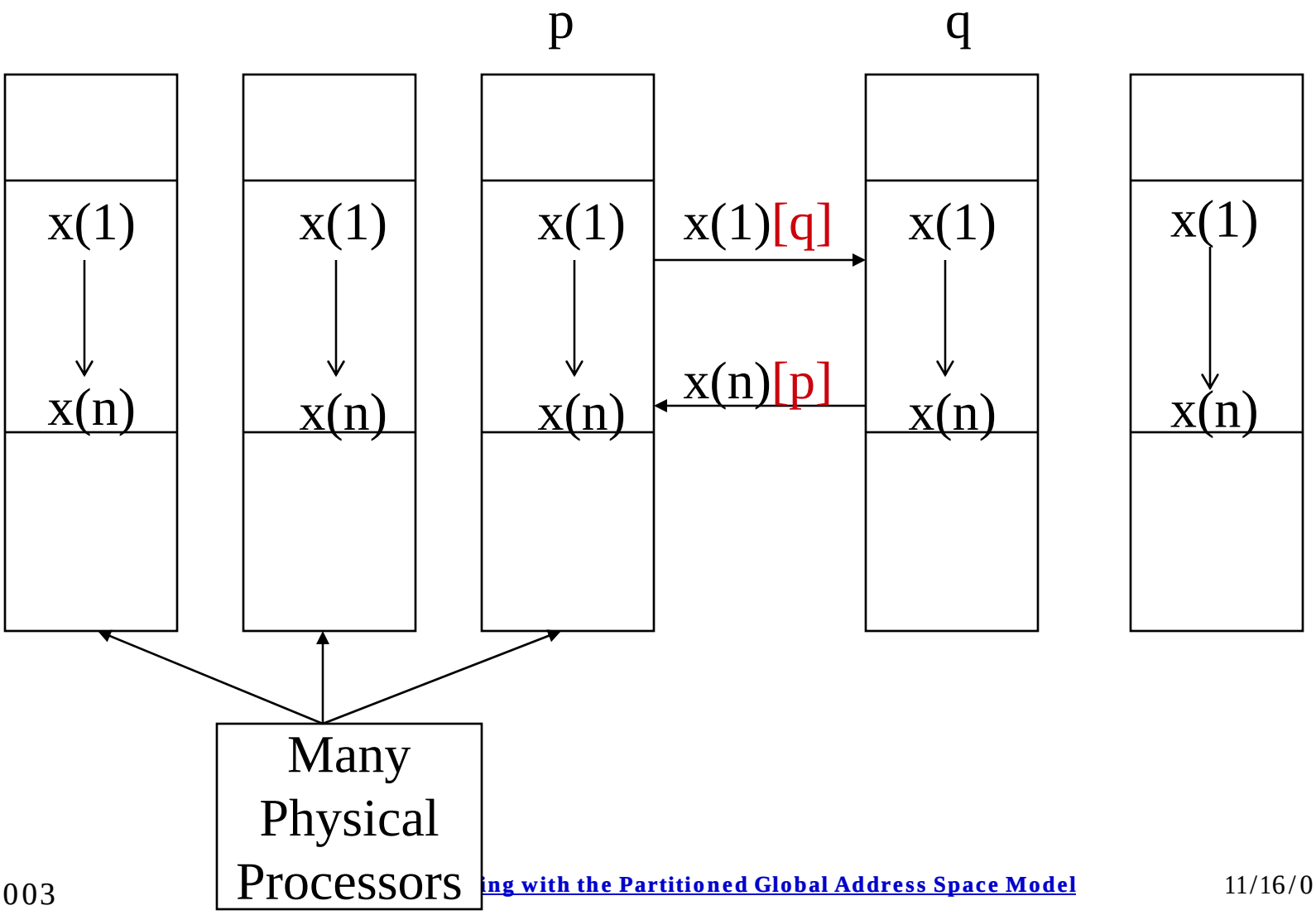
Many-to-One Execution Model



One-to-Many Execution Model



Many-to-Many Execution Model



Communication Using CAF Syntax

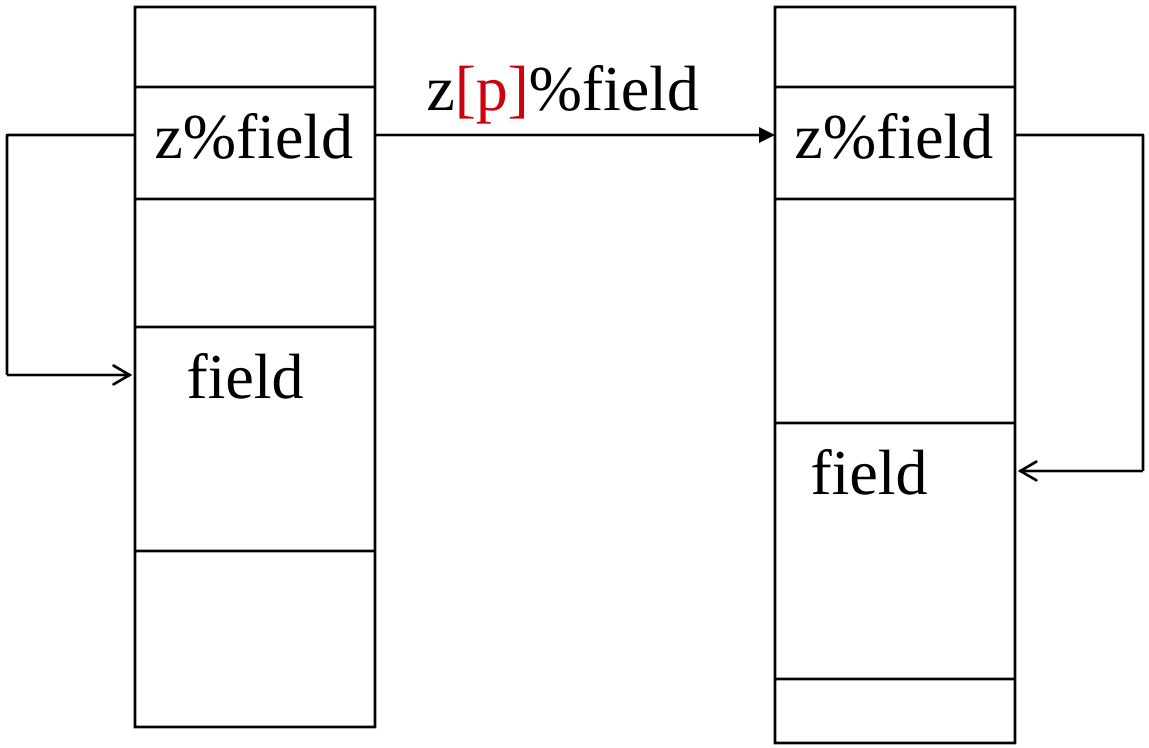
```
y(:) = x(:)[p]
myIndex(:) = index(:)
yourIndex(:) = index(:)[you]
x(index(:)) = y[index(:)]

x(:)[q] = x(:) + x(:)[p]
```

Absent co-dimension defaults to the local object.

Non-Aligned Variables

```
real,allocatable,target :: field (:)
type(field) :: z[*]
allocate(field(0:n+1))
me = this_image(z)
z%field => field
field(0)     = z[me-1]%field(n)
field(n+1) = z[me+1]%field(1)
```

Co-Array Alias to a Remote Field



5. Relative Image Indices

What Do Co-Dimensions Mean?

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

1. Replicate an array of length n, one on each image.
2. Build a map so each image knows how to find the array on any other image.
3. Organize images in a logical (not physical) three dimensional grid.
4. The last co-dimension acts like an assumed size array: *
   ⇒ num_images()/(pxq)
5. A specific implementation could choose to represent memory hierarchy through the co-dimensions.

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/)

6. CAF Intrinsic Procedures

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

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 after all images have allocated

  deallocate(x)

  implied sync before any image deallocates

- Pointers are not allowed to be co-arrays

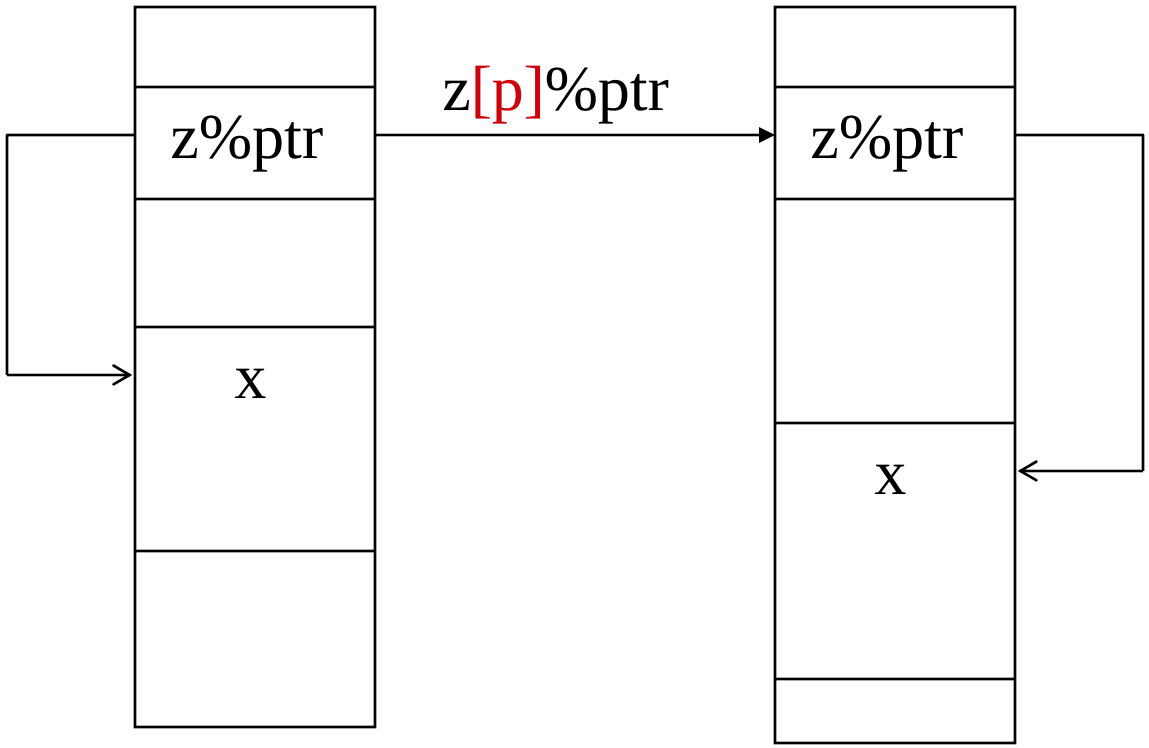
User Defined Derived Types

- 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



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
```

8. 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]
```

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.
- But it contains some features that can be used to emulate object-oriented programming methods.
  - Allocate/deallocate for dynamic memory management
  - Named derived types are similar to classes without methods.
  - Modules can be used to associate methods loosely with objects.
  - Constructors and destructors can be defined to encapsulate parallel data structures.
  - Generic interfaces can be used to overload procedures based on the named types of the actual arguments.

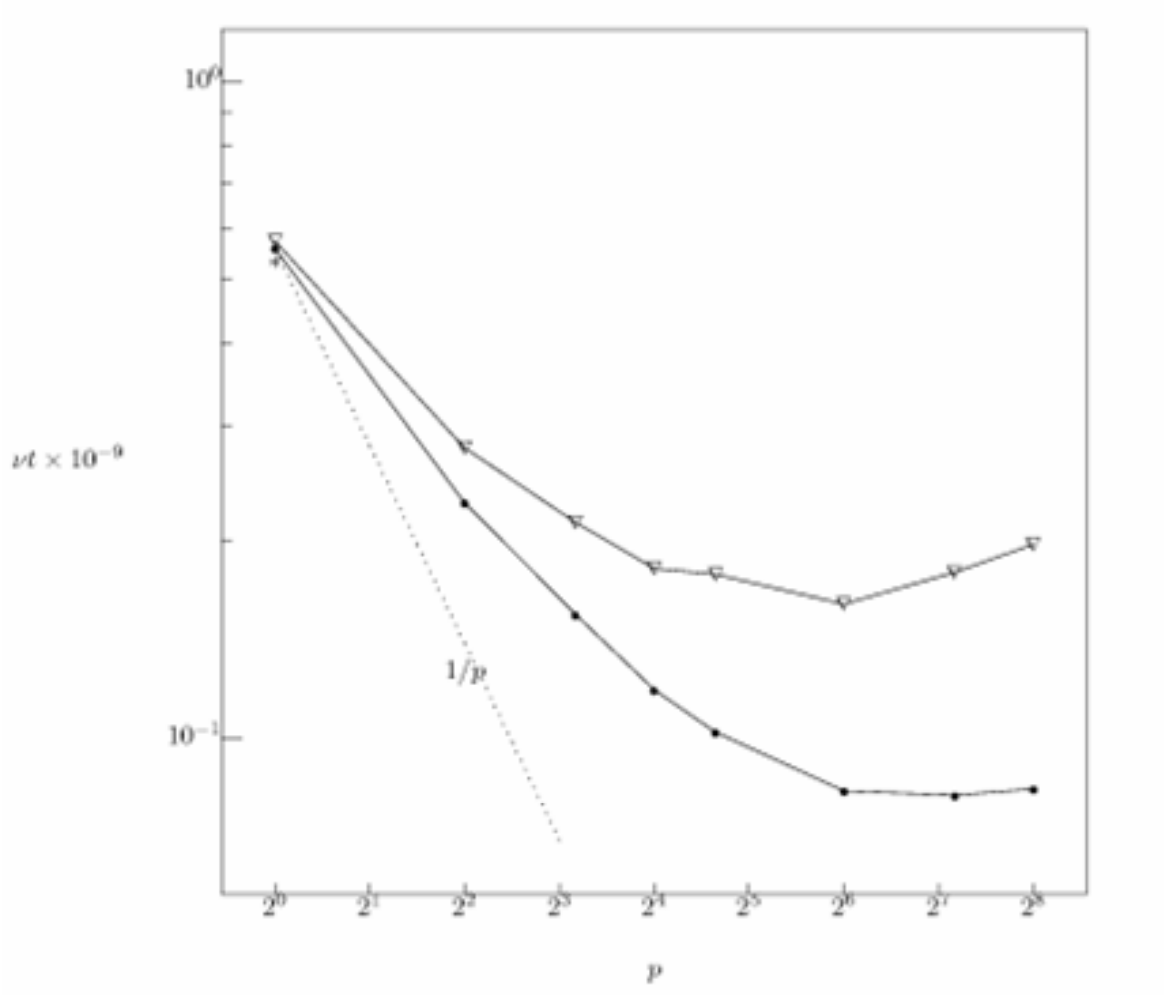
A Parallel “Class Library” for CAF

- Combine the object-based features of Fortran 95 with co-array syntax to obtain an efficient parallel numerical class library that scales to large numbers of processors.
- Encapsulate all the hard stuff in modules using named objects, constructors,destructors, generic interfaces, dynamic memory management.

CAF Parallel "Class Libraries"

```
use BlockMatrices
use BlockVectors

type(PivotVector)  :: pivot[p,*]
type(BlockMatrix) :: a[p,*]
type(BlockVector) :: x[*]

call newBlockMatrix(a,n,p)
call newPivotVector(pivot,a)
call newBlockVector(x,n)
call luDecomp(a,pivot)
call solve(a,x,pivot)
```

LU Decomposition



CAF I/O for Named Objects

```
use BlockMatrices
use DiskFiles

type(PivotVector)  :: pivot[p,*]
type(BlockMatrix) :: a[p,*]
type(DirectAccessDiskFile) :: file

call newBlockMatrix(a,n,p)
call newPivotVector(pivot,a)
call newDiskFile(file)
call readBlockMatrix(a,file)
call luDecomp(a,pivot)
call writeBlockMatrix(a,file)
```

10. Summary

Why Language Extensions?

- Programmer uses a familiar language.
- Syntax gives the programmer control and flexibility.
- Compiler concentrates on local code optimization.
- Compiler evolves as the hardware evolves.
  - Lowest latency and highest bandwidth allowed by the hardware
  - Data ends up in registers or cache not in memory
  - Arbitrary communication patterns
  - Communication along multiple channels

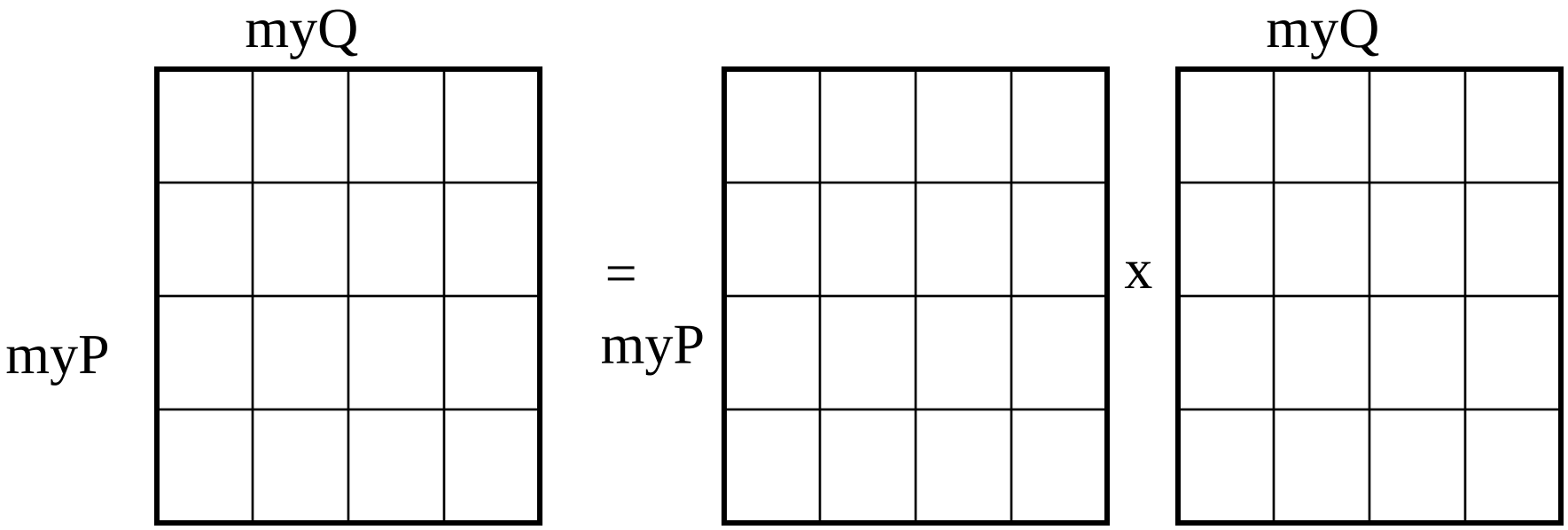
Summary

- 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.

11. Examples

Examples from Linear Algebra

Matrix Multiplication



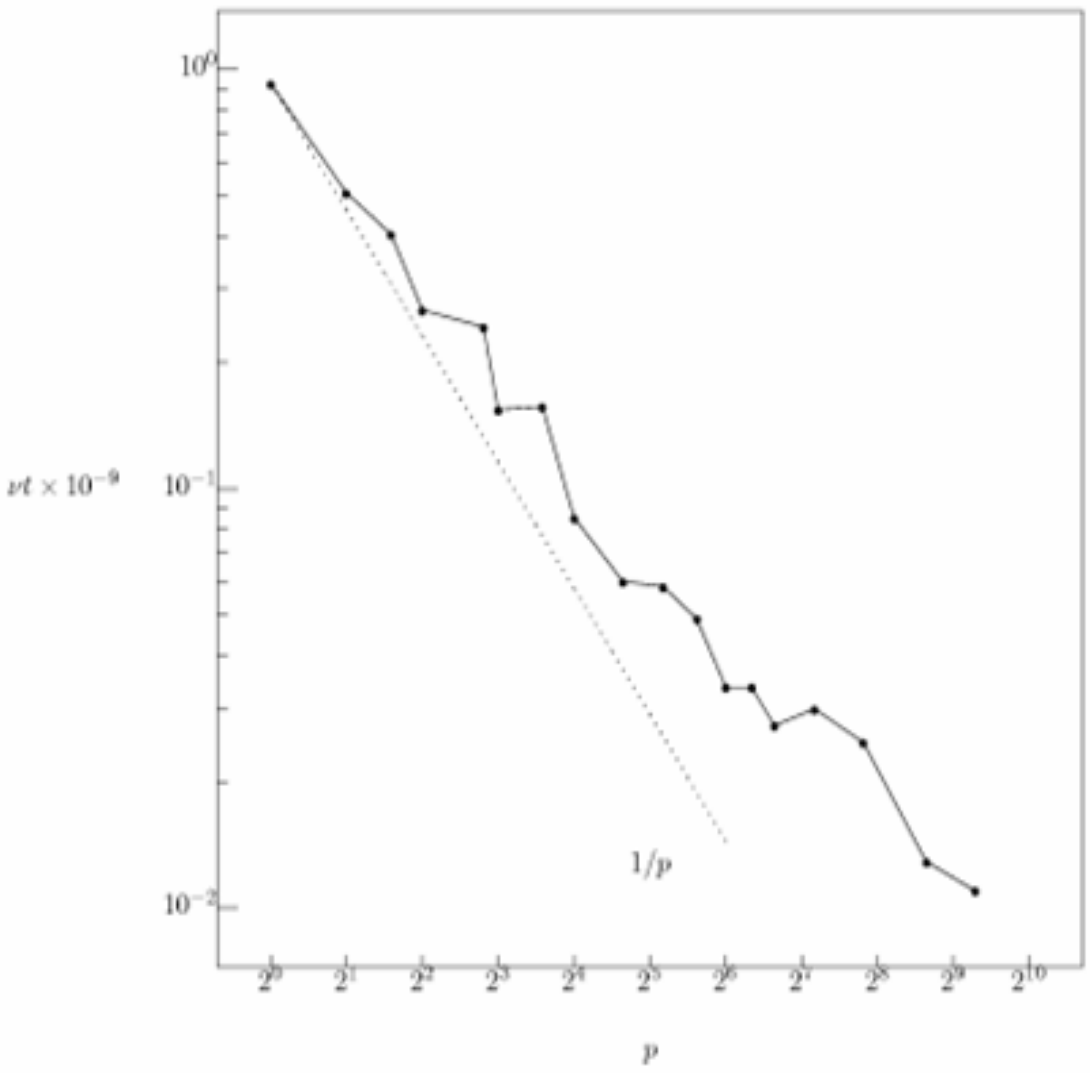
Matrix Multiplication

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

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

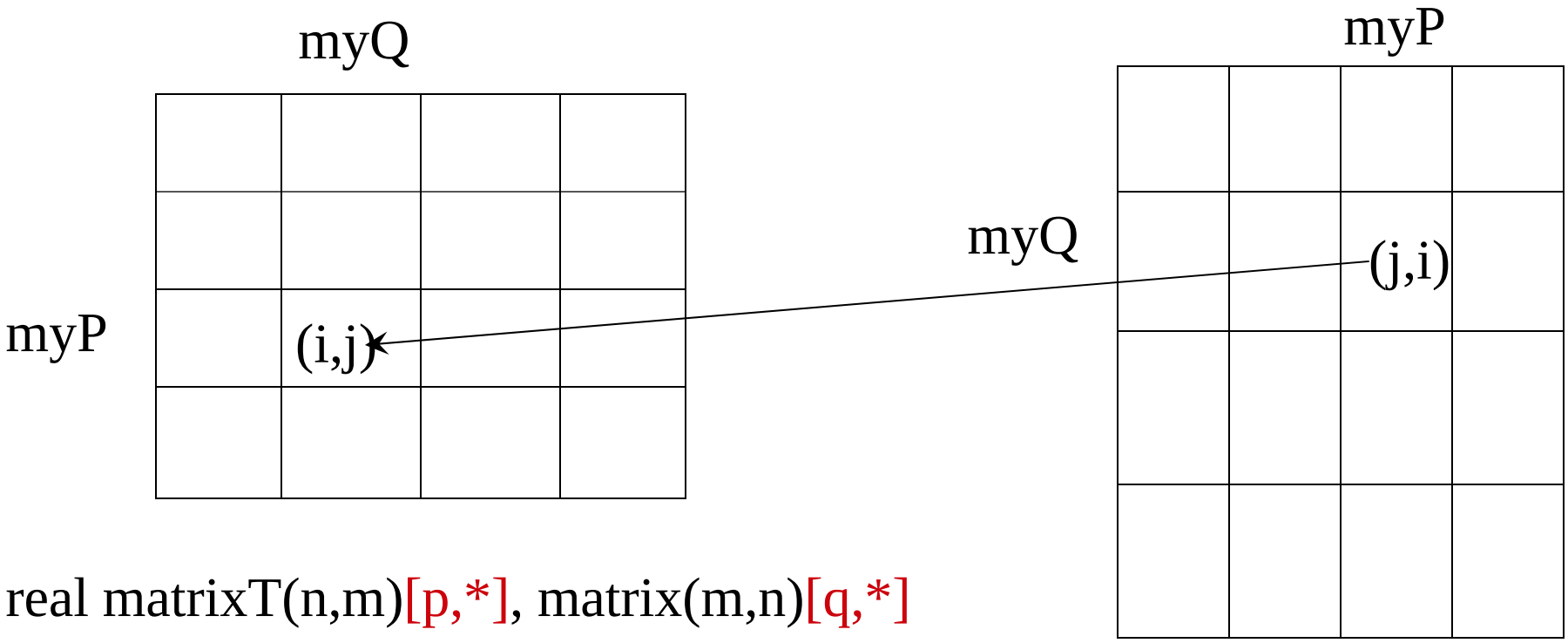
Matrix Multiplication

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

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

Block Matrix Multiplication



Distributed Transpose (1)



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

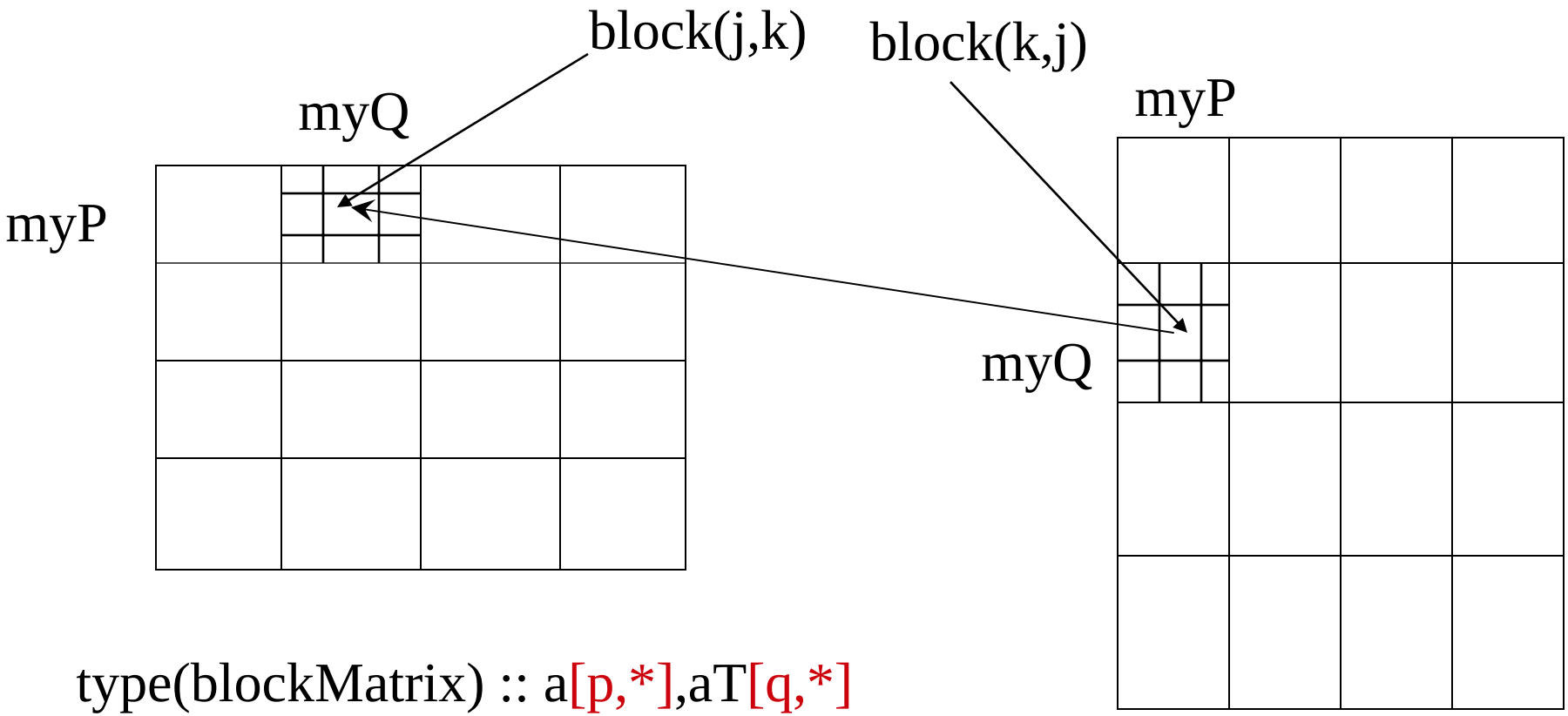
Blocked Matrices (1)

```
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)

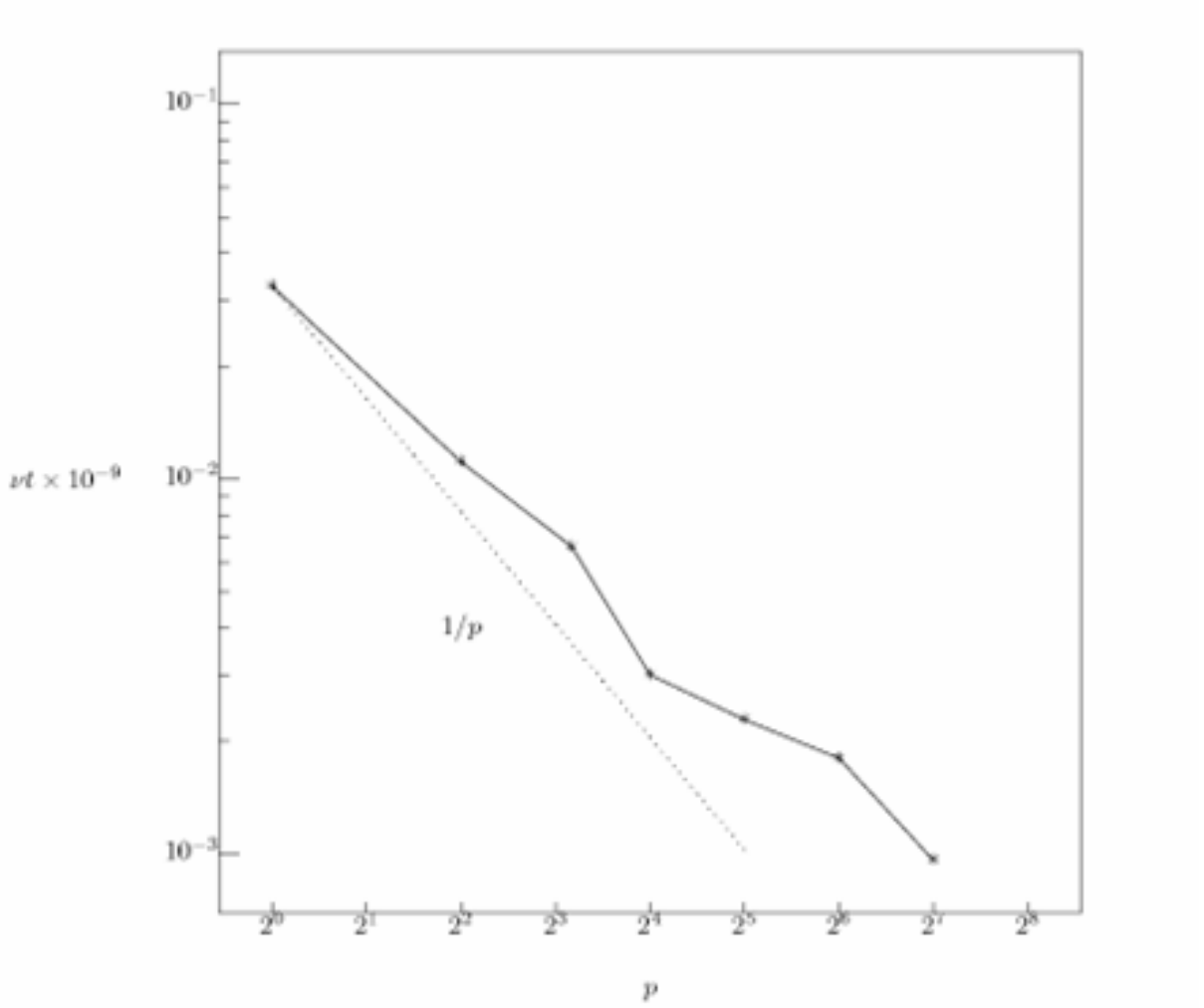
```
type(blockMatrix),allocatable :: a[:,:]
allocate(a[p,*])
allocate(a%block(nRowBlks,nColBlks))
a%block(j,k)%rowSize = nRows
a%block(j,k)%colSize = nCols
```

Distributed Transpose (2)

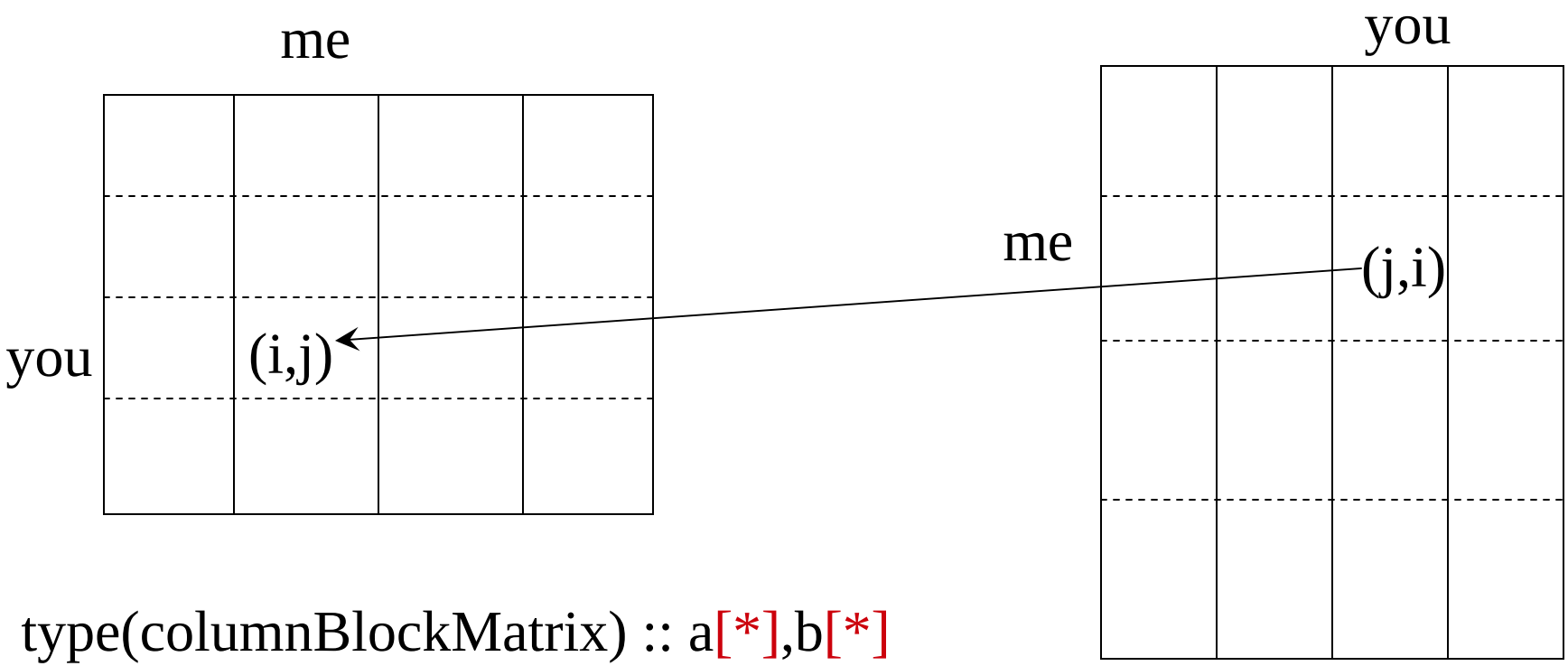


type(blockMatrix) :: a[p,*],aT[q,*]
aT%block(j,k)%element(i,j) = a[myQ,myP]%block(k,j)%element(j,i)

Block Matrix Transpose



Distributed Transpose (3)

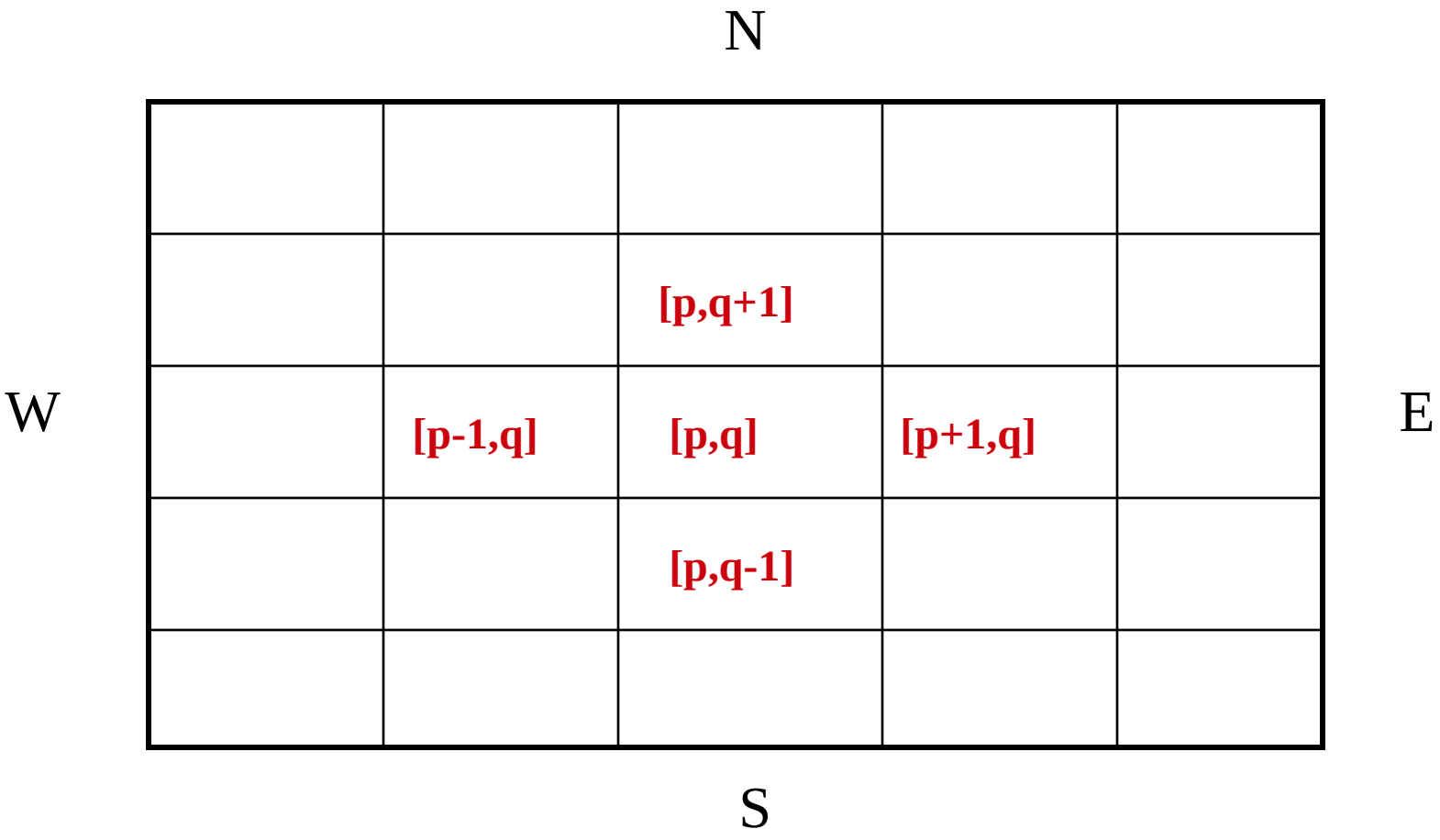


type(columnBlockMatrix) :: a[*],b[*]

a[me]%block(you)%element(i,j) = b[you]%block(me)%element(j,i)

Example from the UK Met Office

Problem Decomposition and Co-Dimensions



Cyclic Boundary Conditions in East-West Directions

```
real,dimension [p,*] :: z
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.**

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)

CRAY Co-Array Fortran

- CAF has been a supported feature of Cray Fortran 90 since release 3.1
- CRAY T3E
  - f90 -Z src.f90
  - mpprun -n7 a.out
- CRAY X1
  - ftn -Z src.f90
  - aprun -n7 a.out

Co-Array Fortran on Other Platforms

- Rice University is developing a source-to-source preprocessor for CAF.
  - www.pmodels.org
- DARPA High Productivity Computing Systems (HPCS) Project wants CAF.
  - IBM, CRAY, SUN
- Open source CAF compiler under consideration by DoE.

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
  - www.pmodels.org

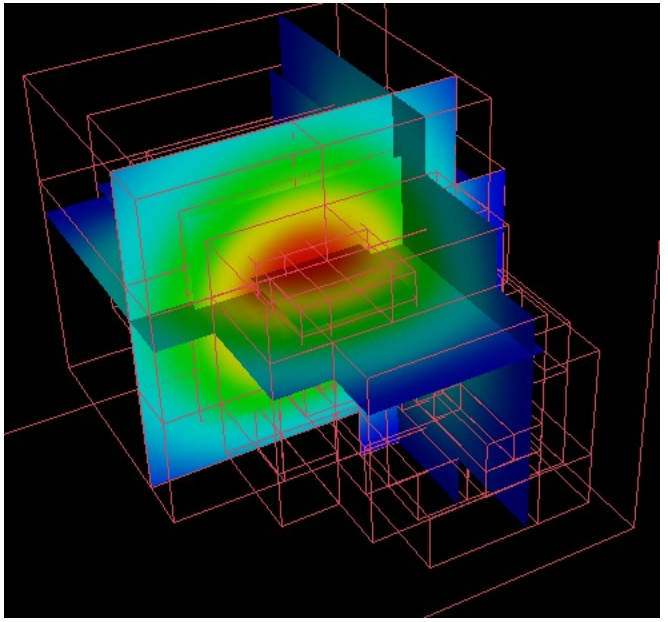
Titanium: A Java Dialect for High Performance Computing

Katherine Yelick

U.C. Berkeley and LBNL

Motivation: 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
  - Structured grids
  - Locally-structured grids (AMR)
  - Unstructured grids (in progress)



Source: J. Bell, LBNL

Titanium Background

- Based on Java, a cleaner C++
  - Classes, automatic memory management, etc.
  - Compiled to C and then machine code, no JVM
- Same parallelism model at UPC and CAF
  - SPMD parallelism
  - Dynamic Java threads are not supported
- Optimizing compiler
  - Analyzes global synchronization
  - Optimizes pointers, communication, memory

Summary of Features Added to Java

- Multidimensional arrays: iterators, subarrays, copying
- Immutable ("value") classes
- Templates
- Operator overloading
- Scalable SPMD parallelism replaces threads
- Global address space with local/global reference distinction
- Checked global synchronization
- Zone-based memory management (regions)
- Libraries for collective communication, distributed arrays, bulk I/O, performance profiling

Outline

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

SPMD Execution Model

- Titanium has the same execution model as UPC and CAF
- Basic Java programs may be run as Titanium programs, 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()
                         + " out of "
                         + Ti.numProcs());
    }
}
```

- Global synchronization done using `Ti.barrier()`

Barriers and Single

- Common source of bugs is barriers or other collective 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 is optional, but useful in understanding compiler messages
- Compiler proves that all processors call barriers together

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 will have the same value after the broadcast.

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 must contain only single variables
- Compiler analysis statically enforces freedom from deadlocks due to barrier and other collectives being called non-collectively "Barrier Inference" [Gay & Aiken]

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 remote particles, compute forces on mine
    Ti.barrier();
    write to my particles using new forces
    Ti.barrier();
  }
 }
}
```

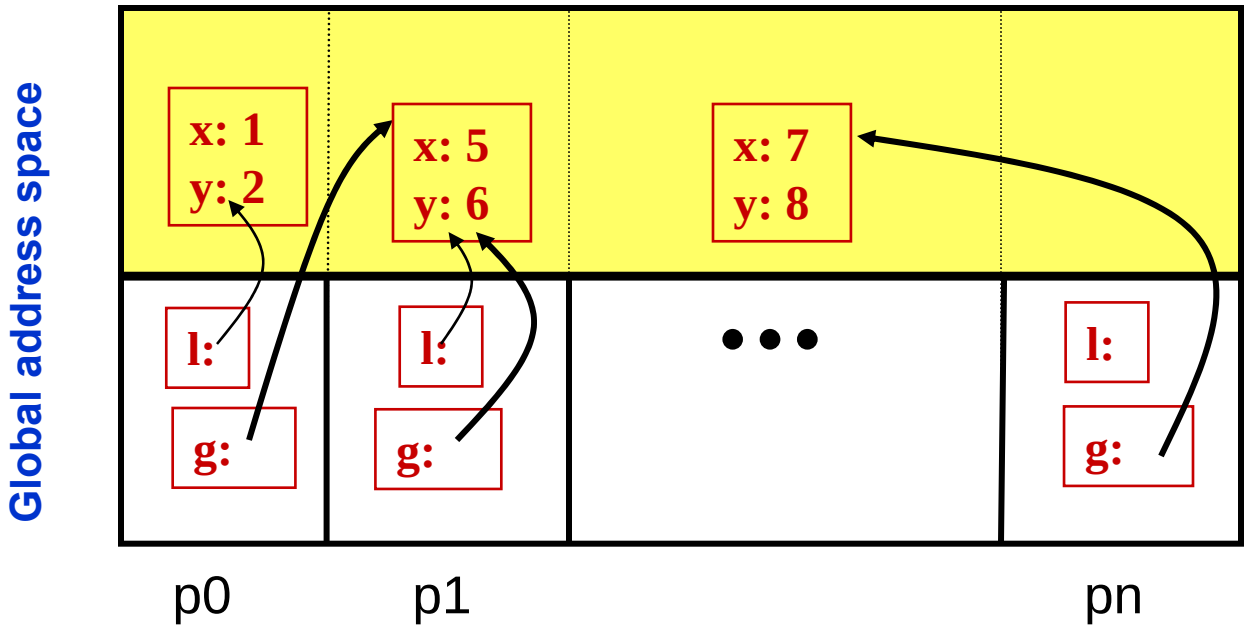
- **Single methods inferred by the compiler**

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
- Compiler/Language Status

Global Address Space

- Globally shared address space is partitioned
- References (pointers) are either local or global (meaning possibly remote)



Object heaps are shared

Program stacks are private

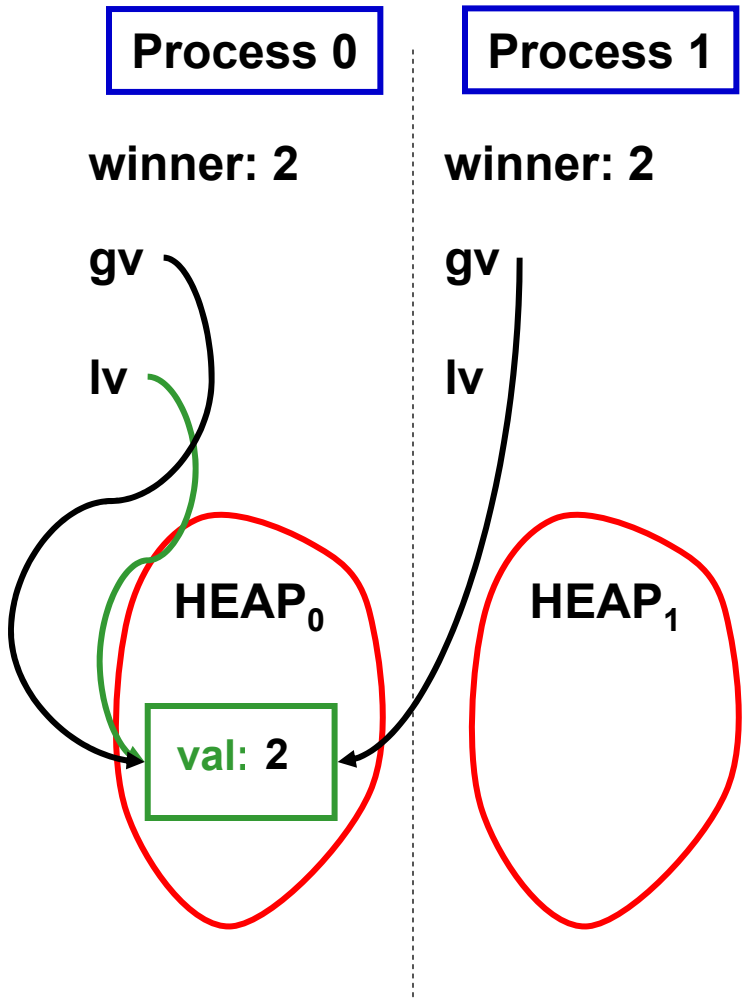
Use of Global / Local

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

Global Address Space

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

```
class C { public int val;... }
C gv;        // global pointer
C local lv; // local pointer
if (Ti.thisProc() == 0) {
      lv = new C();
}
gv = broadcast lv from 0;
//data race
gv.val = Ti.thisProc()+1;

int winner = gv.val
```



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] a;

a = new int [1:100];

a[1] = 2*a[1] - a[0] – a[2];
```

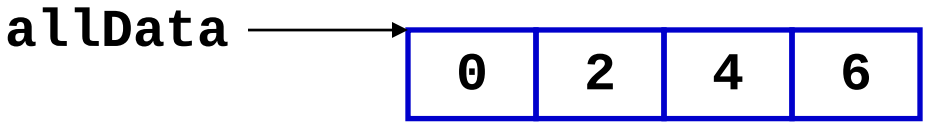
- Will discuss these more later…

Explicit Communication: Exchange

- To create shared data structures
  - each processor builds its own piece
  - pieces are exchanged (for objects, 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:



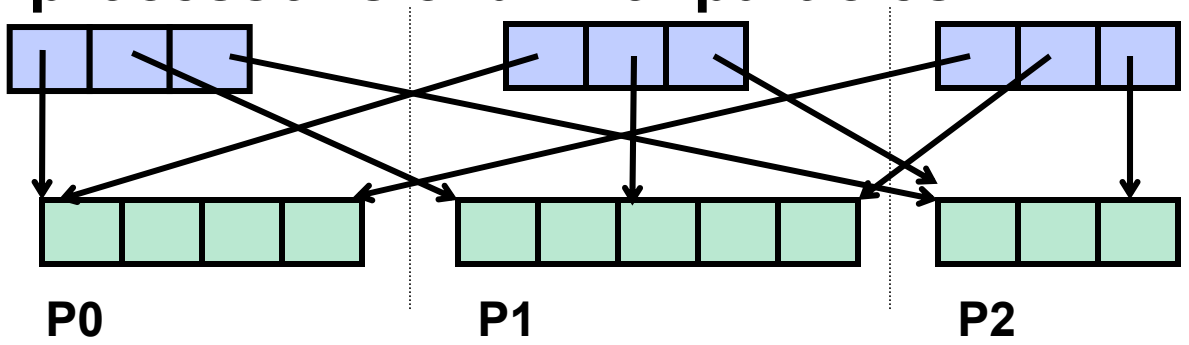
Distributed Data Structures

- **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);
```

All to all broadcast

- **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:
  - Has a reputation of slowing serial code
  - Does not scale well in a parallel environment
- Titanium approach:
  - Preserves safety – cannot deallocate live data
  - Garbage collection is the default (on most platforms)
  - Higher performance is possible using region-based explicit memory management
  - Takes advantage of memory management phases

Region-Based Memory Management

- Need to organize data structures
- Allocate set of objects (safely)
- Delete them with a single explicit call (fast)

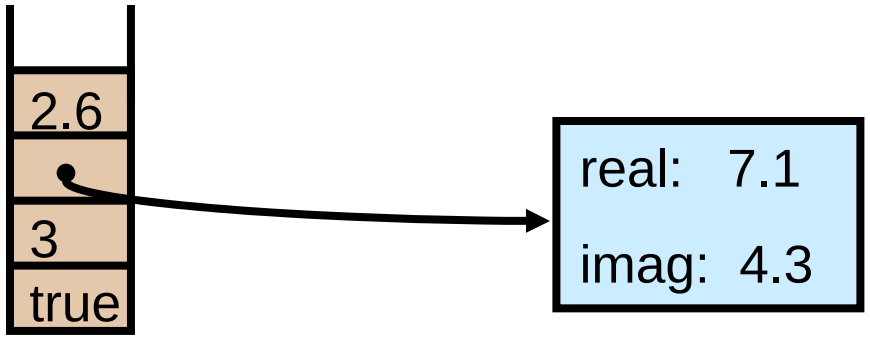
```
PrivateRegion r = new PrivateRegion();
for (int j = 0; j < 10; j++) {
   int[] x = new ( r ) int[j + 1];
   work(j, x);
}
try { r.delete(); }
 catch (RegionInUse oops) {
    System.out.println("failed to delete");
 }
}
```

Outline

- Titanium Execution Model
- Titanium Memory Model
- Support for Serial Programming
  - Immutables
  - Operator overloading
  - Multidimensional arrays
  - Templates
- Performance and Applications
- Compiler/Language Status

Java Objects

- Primitive scalar types: boolean, double, int, etc.
  - implementations store these on the program stack
  - access is fast -- comparable to other languages
- Objects: user-defined and standard library
  - always allocated dynamically in the heap
  - passed by pointer value (object sharing)
  - has implicit level of indirection
  - 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; }
}
```

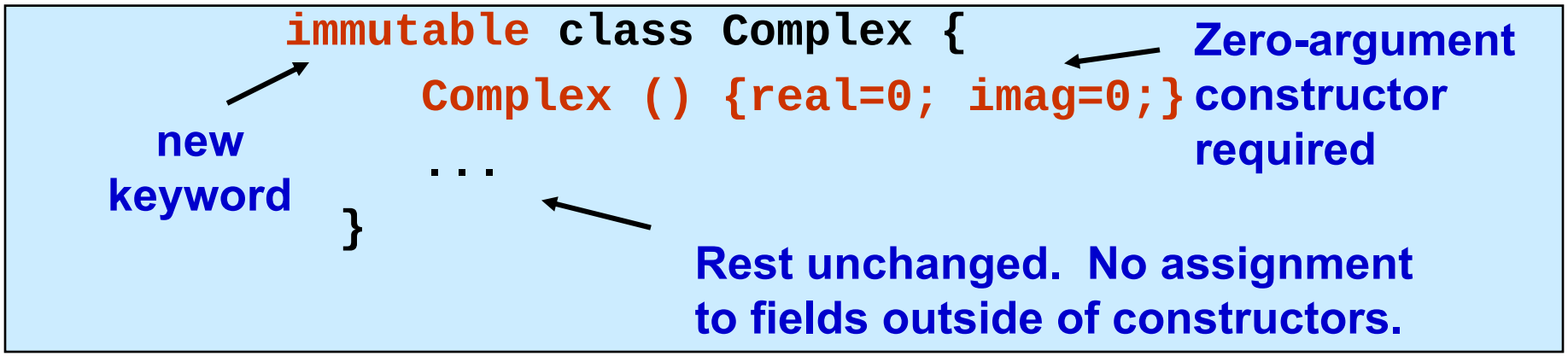
```
Complex c = new Complex(7.1, 4.3);
c = c.add(c);
class VisComplex extends Complex { ... }
```

Immutable Classes in Titanium

- For small objects, would sometimes prefer
  - to avoid level of indirection and allocation overhead
  - pass by value (copying of entire object)
  - especially when immutable -- fields never modified
    - extends the idea of primitive values to user-defined types
- Titanium introduces immutable classes
  - all fields are implicitly final (constant)
  - cannot inherit from or be inherited by other classes
  - needs to have 0-argument constructor
- Examples: Complex, xyz components of a force
- Note: considering lang. extension to allow mutation

Example of Immutable Classes

- The immutable complex class nearly the same



- 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);
```

- Addresses performance and programmability
  - Similar to C structs in terms of performance
  - Support for Complex with a general mechanism

Operator Overloading

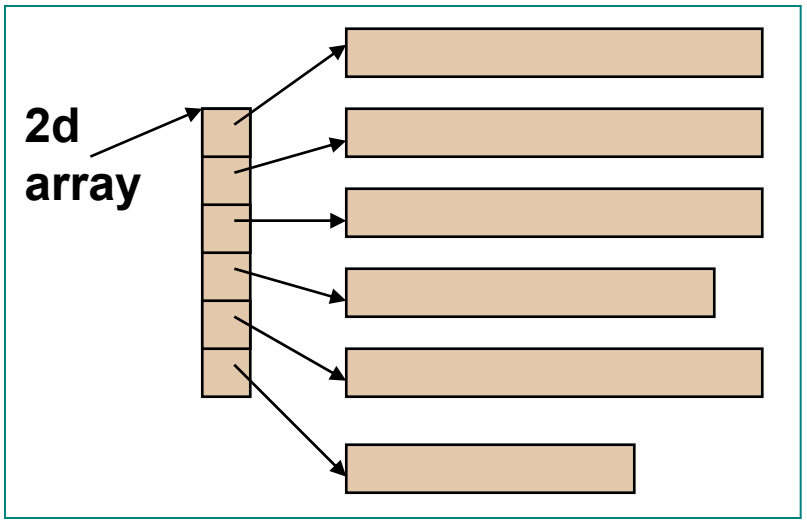
- Titanium provides operator overloading
  - Convenient in scientific code
  - Feature is similar to that in C++

```
class Complex {
  ...
  public Complex op+(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;
```

Arrays in Java

- Arrays in Java are objects
- Only 1D arrays are directly supported
- Multidimensional arrays are arrays of arrays
- General, but 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
  - One array may be a subarray of another
    - e.g., a is interior of b, or a is all even elements of b
    - can easily refer to rows, columns, slabs or boundary regions as sub-arrays of a larger array
  - Indexed by Points (tuples of ints)
  - Built on a rectangular set of Points, RectDomain
  - Points, Domains and RectDomains are built-in immutable classes, with useful literal syntax
- Support for AMR and other grid computations
  - domain operations: intersection, shrink, border
  - bounds-checking can be disabled after debugging

Unordered Iteration

- Motivation:
  - Memory hierarchy optimizations are essential
  - Compilers sometimes do these, but hard in general
- Titanium has explicitly unordered iteration
  - Helps the compiler with analysis
  - Helps programmer avoid indexing details

```
foreach (p in r) { … A[p] … }
```

    - p is a Point (tuple of ints), can be used as array index
    - r is a RectDomain or Domain
- Additional operations on domains to transform
- 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 optional stride)

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

- Array declared by num 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];

foreach(p in c.domain()) { c[p] = a[p] + b[p]; }
```

No array allocation here

Syntactic sugar

Optional stride

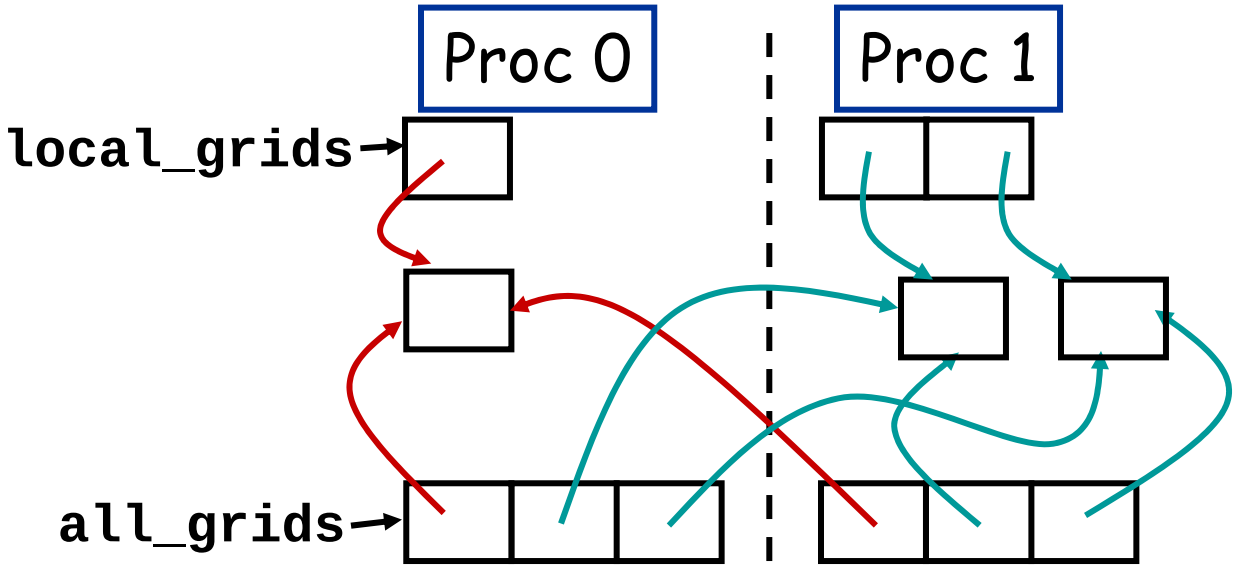
Equivalent loops

MatMul with Titanium Arrays

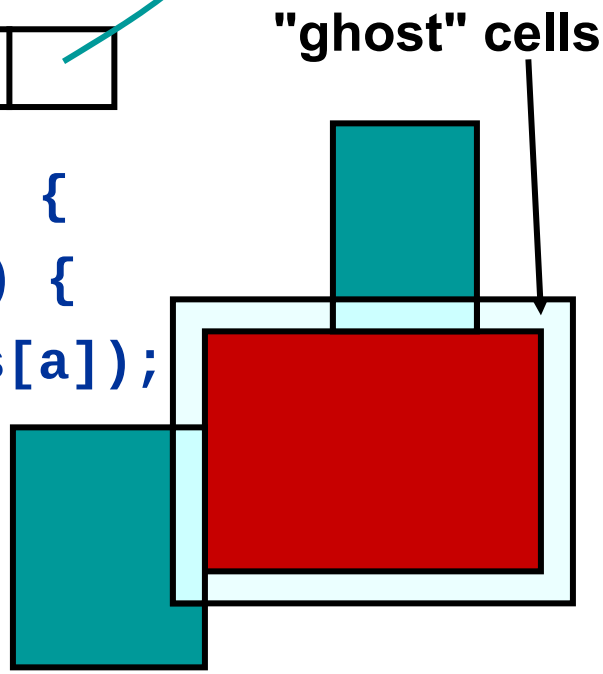
```
public static void matMul(double [2d] a,
                          double [2d] b,
                          double [2d] c) {
  foreach (ij in c.domain()) {
    double [1d] aRowi = a.slice(1, ij[1]);
    double [1d] bColj = b.slice(2, ij[2]);
    foreach (k in aRowi.domain()) {
      c[ij] += aRowi[k]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

Current performance: comparable to 3 nested loops in C

Example: Setting Boundary Conditions




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



Templates

- Many applications use containers:
  - Parameterized by dimensions, element types,…
  - Java supports parameterization through inheritance
    - Can only put Object types into containers
    - Inefficient when used extensively
- Titanium provides a template mechanism closer to C++
  - Can be instantiated with non-object types (double, Complex) as well as objects
- Example: Used to build a distributed array package
  - Hides the details of exchange, indirection within the data structure, etc.

Example of Templates

```
template <class Element> class Stack {
    . . .
    public Element pop() {...}
    public void push( Element arrival ) {...}
}

template Stack<int> list = new template Stack<int>();
list.push( 1 );
int x = list.pop();
```

Not an object

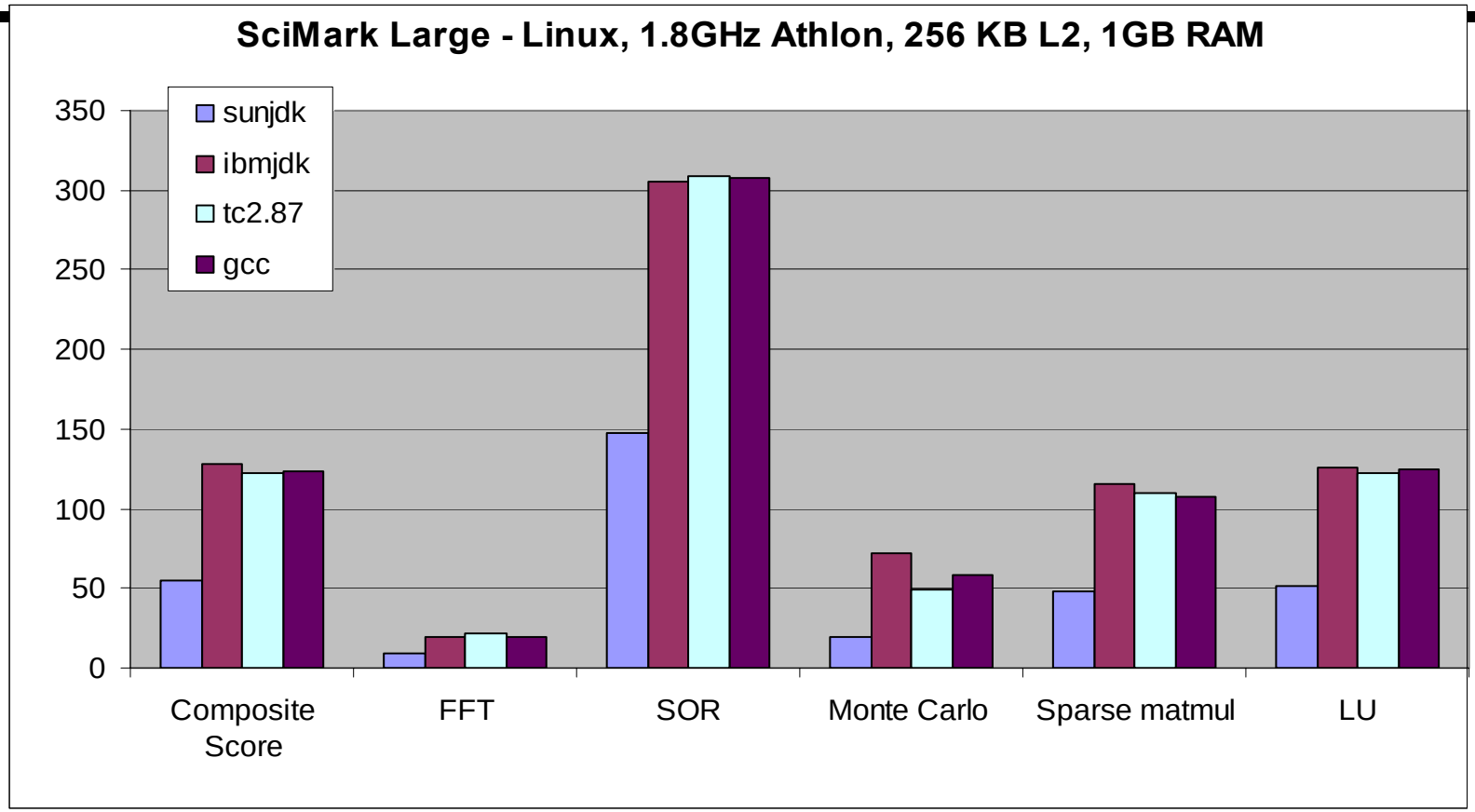
Strongly typed,
No dynamic cast

- Addresses programmability and performance

Outline

- Titanium Execution Model
- Titanium Memory Model
- Support for Serial Programming
- Performance and Applications
  - Serial Performance on pure Java (SciMark)
  - Parallel Applications
  - Compiler status & usability results
- Compiler/Language Status

Java Compiled by Titanium Compiler



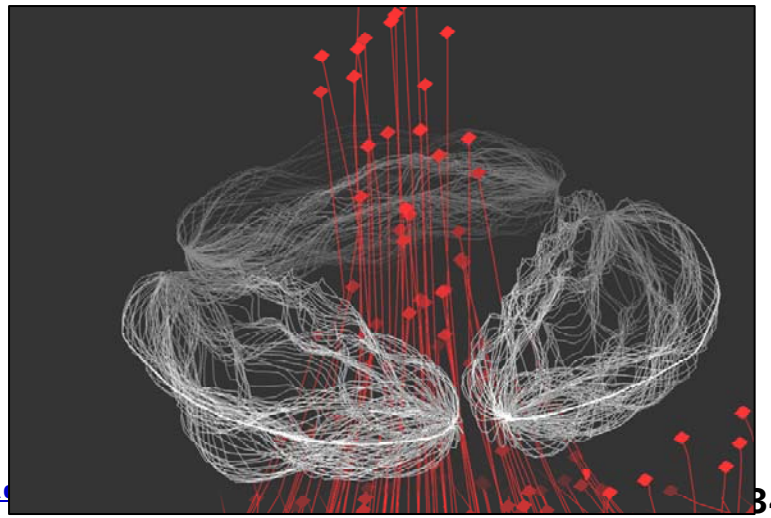
- Sun JDK 1.4.1_01 (HotSpot(TM) Client VM) for Linux
- IBM J2SE 1.4.0 (Classic VM cxia32140-20020917a, jitc JIT) for 32-bit Linux
- Titaniumc v2.87 for Linux, gcc 3.2 as backend compiler -O3. no bounds check
- gcc 3.2, -O3 (ANSI-C version of the SciMark2 benchmark)

Applications in Titanium

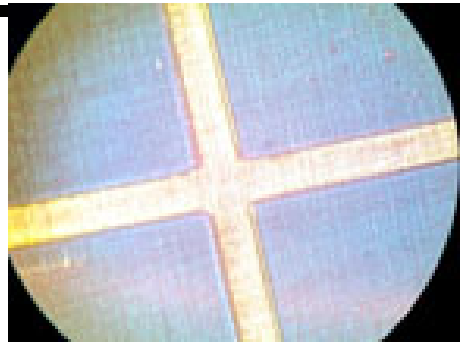
- Benchmarks and Kernels
  - Scalable Poisson solver for infinite domains
  - NAS PB: MG, FT, IS, CG
  - Unstructured mesh kernel: EM3D
  - Dense linear algebra: LU, MatMul
  - Tree-structured n-body code
  - Finite element benchmark
- Larger applications
  - Gas Dynamics with AMR
  - Heart and Cochlea simulation (ongoing)
  - Genetics: micro-array selection
  - Ocean modeling with AMR (in progress)

Heart Simulation: Immersed Boundary Method

- Problem: compute blood flow in the heart
  - Modeled as an elastic structure in an incompressible fluid.
    - The "immersed boundary method" [Peskin and McQueen].
    - 20 years of development in model
  - Many other applications: blood clotting, inner ear, paper making, embryo growth, and more
- Can be used for design of prosthetics
  - Artificial heart valves
  - Cochlear implants



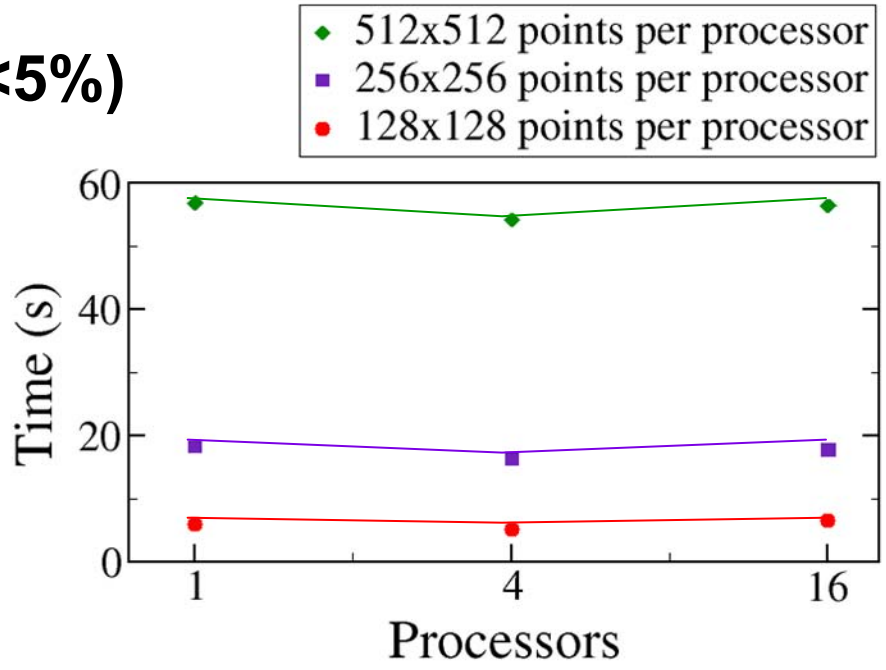
MOOSE Application



- Problem: Genome Microarray construction
  - Used for genetic 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 from a sequenced genome (based on uniqueness and various structural and chemical properties)
  - First parallel implementation for solving this problem
  - Uses dynamic load balancing within Titanium
  - Significant memory and I/O demands for larger genomes

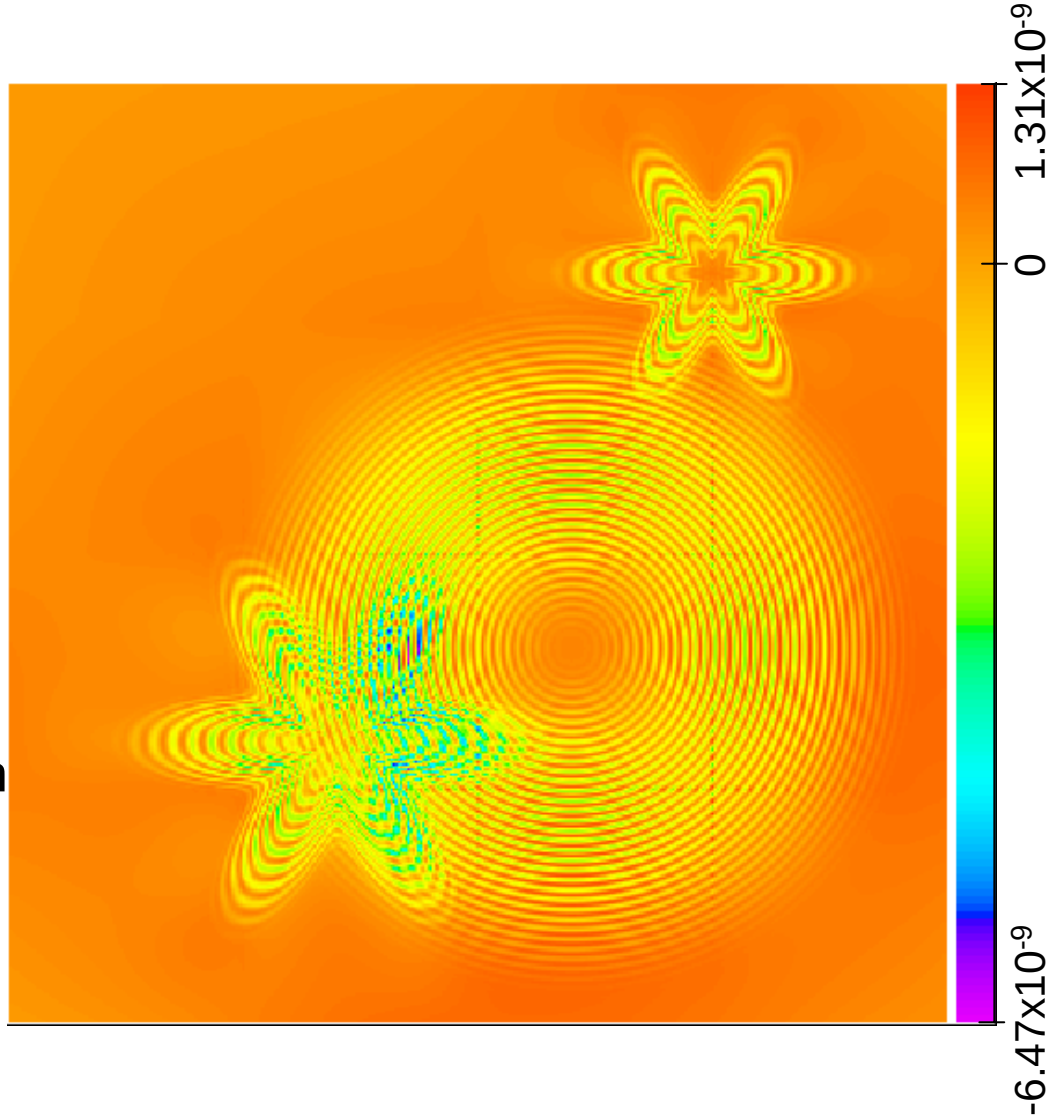
Scalable Parallel Poisson Solver

- 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 (<5%)**
- Performance on
  - **SP2 (shown) and T3E**
  - **scaled speedups**
  - **nearly ideal (flat)**
- Currently 2D and non-adaptive



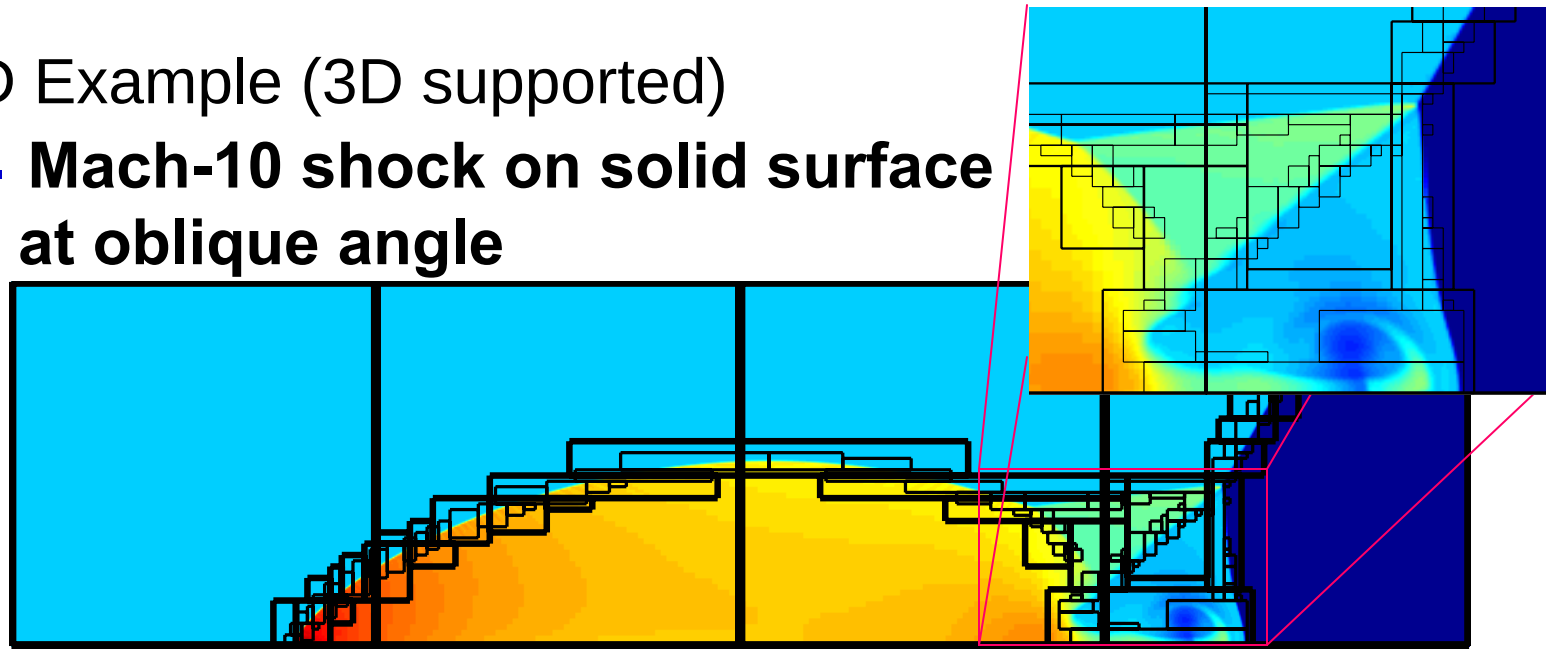
Error on High-Wavenumber Problem

- 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

- Hyperbolic Solver [McCorquodale and Colella]
  - **Implementation of Berger-Colella algorithm**
  - **Mesh generation algorithm included**
- 2D Example (3D supported)
  - **Mach-10 shock on solid surface at oblique angle**



- Future: Self-gravitating gas dynamics package

Outline

- Titanium Execution Model
- Titanium Memory Model
- Support for Serial Programming
- Performance and Applications
- Compiler/Language Status

Titanium Compiler Status

- Titanium runs on almost any machine
  - Requires a C compiler and C++ for the translator
  - Pthreads for shared memory
  - GASNet for distributed memory, which exists on
    - Quadrics, IBM/SP (LAPI), Myrinet (GM), Infiniband, and MPI
    - Shared with Berkeley UPC compiler
- Recent language and compiler work
  - Indexed (scatter/gather) array copy
  - Non-blocking array copy underway
  - Loop level cache optimizations
  - Inspector/Executor underway

Programmability

- Heart simulation developed in ~1 year
  - Extended to support 2D structures for Cochlea model in ~1 month
- Preliminary code length measures
  - Simple torus model
    - Serial Fortran torus code is 17045 lines long (2/3 comments)
    - Parallel Titanium torus version is 3057 lines long.
  - Full heart model
    - Shared memory Fortran heart code is 8187 lines long
    - Parallel Titanium version is 4249 lines long.
  - Need to be analyzed more carefully, but not a significant overhead for distributed memory parallelism

Current Work & Future Plans

- Unified communication layer with UPC: GASNet
- Exploring communication overlap optimizations
  - Explicit (programmer-controlled) and automated
- Analysis and refinement of cache optimizations
- Additional language support for unstructured grids
  - Arrays over general domains
  - Arrays with multiple values per grid point
- Continued work on new and existing applications

http://titanium.cs.berkeley.edu

Titanium Group (Past and Present)

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

- Greg Balls
- Andrew Begel
- Dan Bonachea
- Kaushik Datta
- David Gay
- Ed Givelberg
- Arvind Krishnamurthy

- Ben Liblit
- Peter McQuorquodale (LBNL)
- Sabrina Merchant
- Carleton Miyamoto
- Chang Sun Lin
- Geoff Pike
- Luigi Semenzato (LBNL)
- Armando Solar-Lezama
- Jimmy Su
- Tong Wen (LBNL)
- Siu Man Yau
- and many undergraduate researchers

http://titanium.cs.berkeley.edu

Example of Data Input

- **Reading from keyboard, uses Java exceptions**

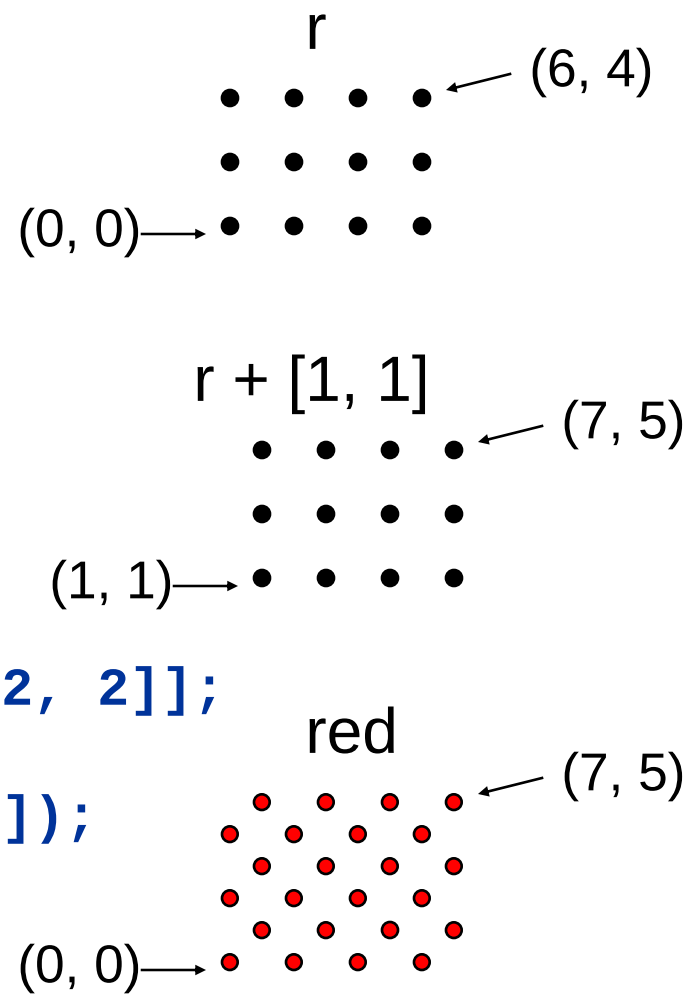
```
int myCount = 0;
int single allCount = 0;
if (Ti.thisProc() == 0)
  try {
     DataInputStream kb =
        new DataInputStream(System.in);
     myCount =
        Integer.valueOf(kb.readLine()).intValue();
  } catch (Exception e) {
    System.err.println("Illegal Input");
  }
allCount = broadcast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

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
  - Although recent compiler analysis infers this distinction and uses it for performing optimizations
- Any object may be referenced by global or local pointers
- 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

Domain Example

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




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

Example using Domains and foreach

- Gauss-Seidel red-black computation in multigrid

```
void gsrb() {
for (Domain<2> d = red; 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];
  }
}
```

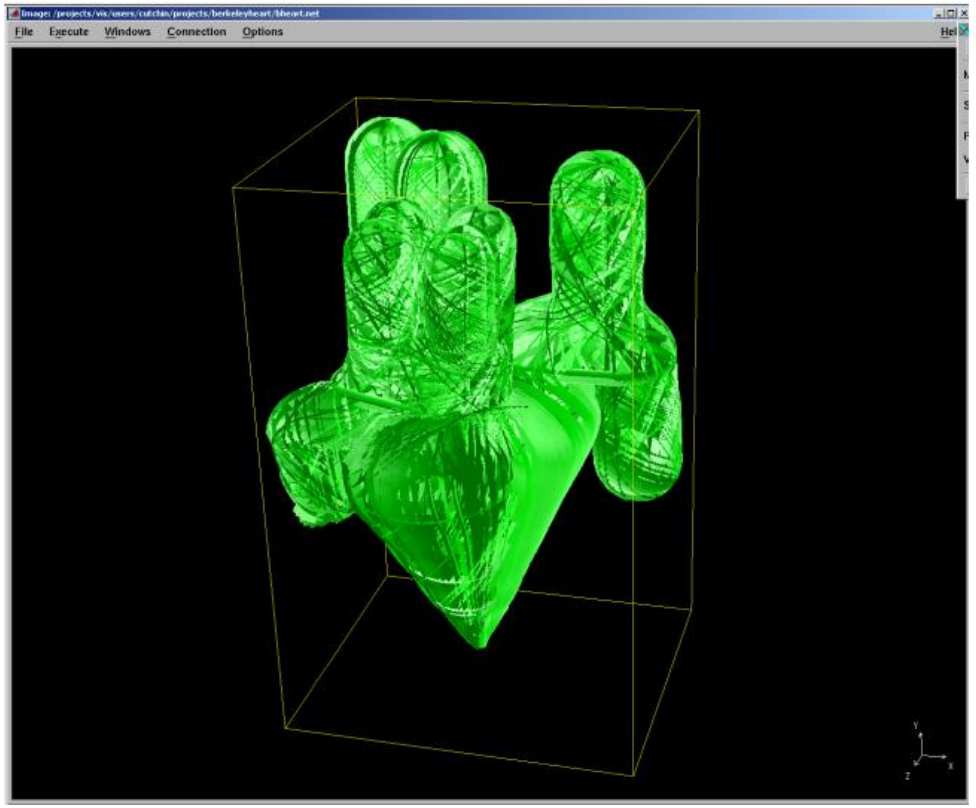
unordered iteration

SciMark Benchmark

- Numerical benchmark for Java, with C versions
  - purely sequential, no Titanium extensions
- Five kernels:
  - FFT (complex, 1D)
  - Successive Over-Relaxation (SOR)
  - Monte Carlo integration (MC)
  - Sparse matrix multiply
  - dense LU factorization
- Results are reported in Mflops
- From Roldan Pozo at NIST
  - http://math.nist.gov/scimark2

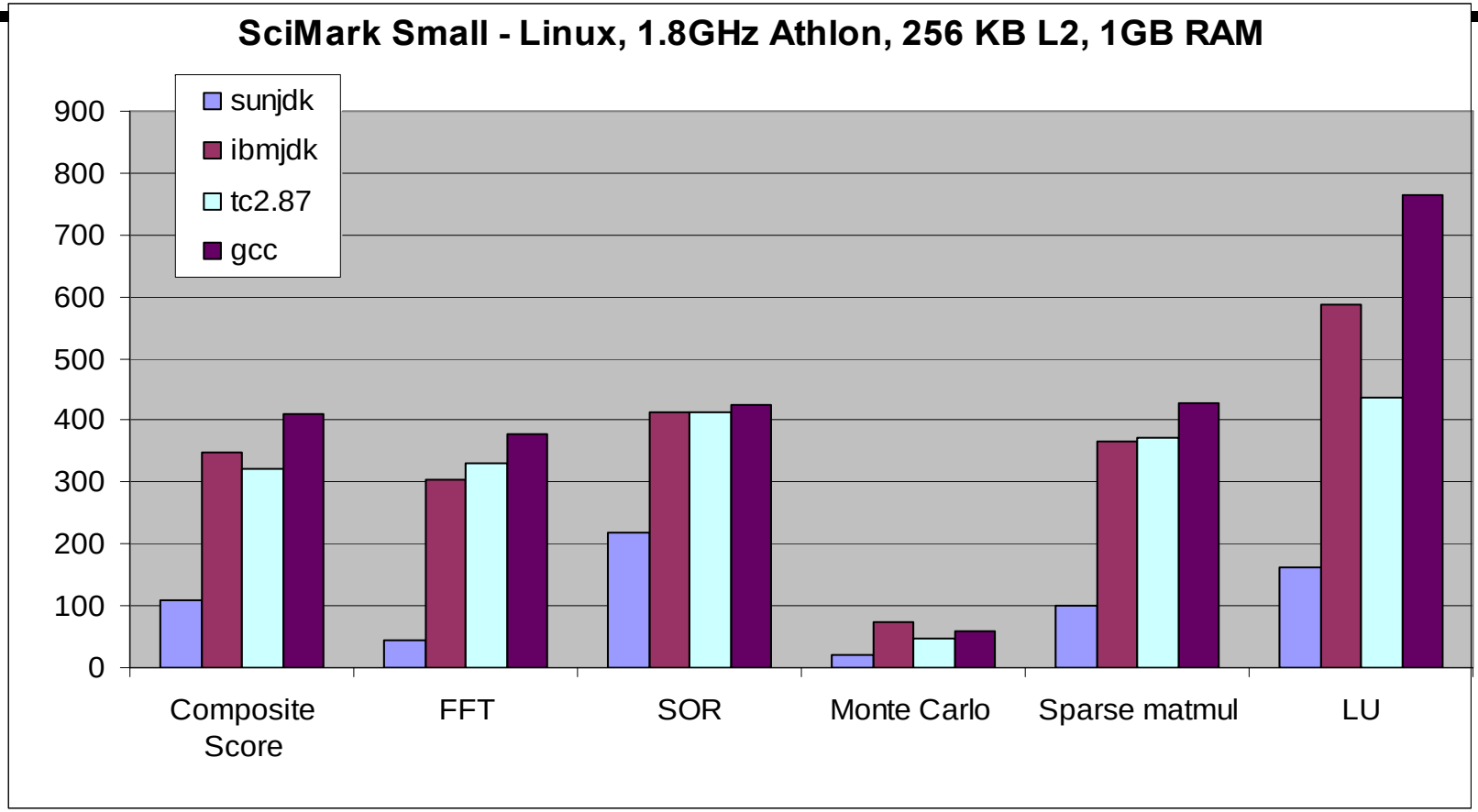
Fluid Flow in Biological Systems

- Immersed Boundary Method
  - Material (e.g., heart muscles, cochlea structure) modeled by grid of material points
  - Fluid space modeled by a regular lattice
- Irregular material points need to interact with regular fluid lattice
  - Trade-off between load balancing of fibers and minimizing communication
  - Memory and communication intensive
  - Includes a Navier-Stokes solver and a 3-D FFT solver



- Heart simulation is complete, Cochlea simulation is close to done
  - First time that immersed boundary simulation has been done on distributed-memory machines
  - Working on a Ti library for doing other immersed boundary simulations

Java Compiled by Titanium Compiler



- Sun JDK 1.4.1_01 (HotSpot(TM) Client VM) for Linux
- IBM J2SE 1.4.0 (Classic VM cxia32140-20020917a, jitc JIT) for 32-bit Linux
- Titaniumc v2.87 for Linux, gcc 3.2 as backend compiler -O3. no bounds check
- gcc 3.2, -O3 (ANSI-C version of the SciMark2 benchmark)

Implementation Portability Status

- Titanium has been tested on:
  - **POSIX-compliant workstations & SMPs**
  - **Clusters of uniprocessors or SMPs**
  - **Cray T3E**
  - **IBM SP**
  - **SGI Origin 2000**
  - **Compaq AlphaServer**
  - **MS Windows/GNU Cygwin**
  - **and others…**

Automatic portability: Titanium applications run on all of these!

Very important productivity feature for debugging & development

- Supports many communication layers
  - **High performance networking layers:**
    - **IBM/LAPI, Myrinet/GM, Quadrics/Elan, Cray/shmem, Infiniband (soon)**
  - **Portable communication layers:**
    - **MPI-1.1, TCP/IP (UDP)**

http://titanium.cs.berkeley.edu

Parallel Programming with the Partitioned Global Address Space Model

Summary

Bill Carlson

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://titanium.cs.berkeley.edu