CS 267
Unified Parallel C (UPC)

Kathy Yelick

http://www.cs.berkeley.edu/~yelick/cs267

Slides adapted from some by Tarek El-Ghazawi (GWU)

UPC Outline

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

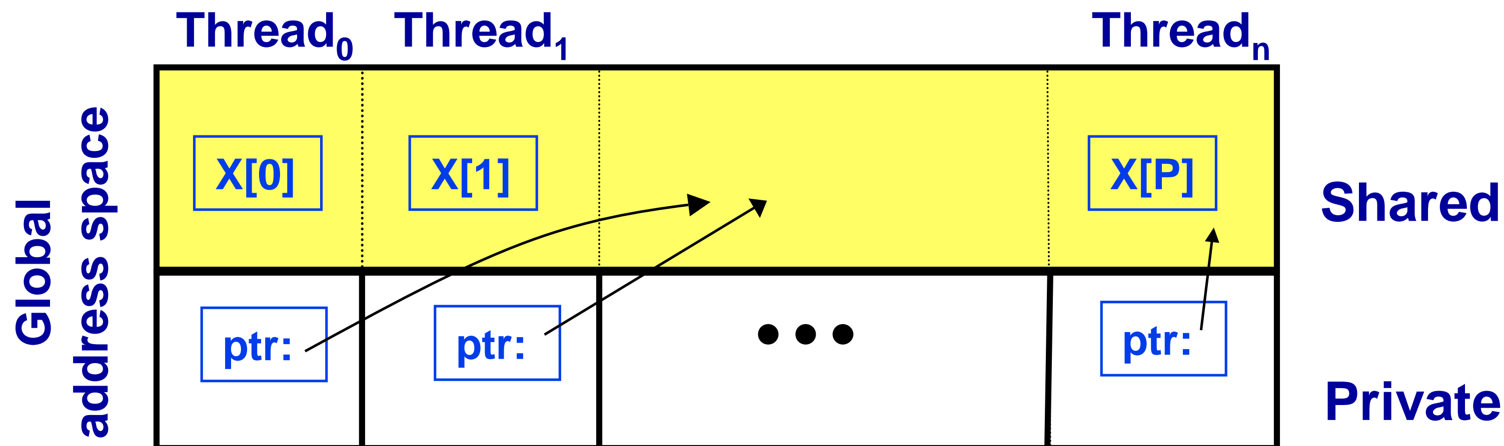
Context

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

Partitioned Global Address Space Languages

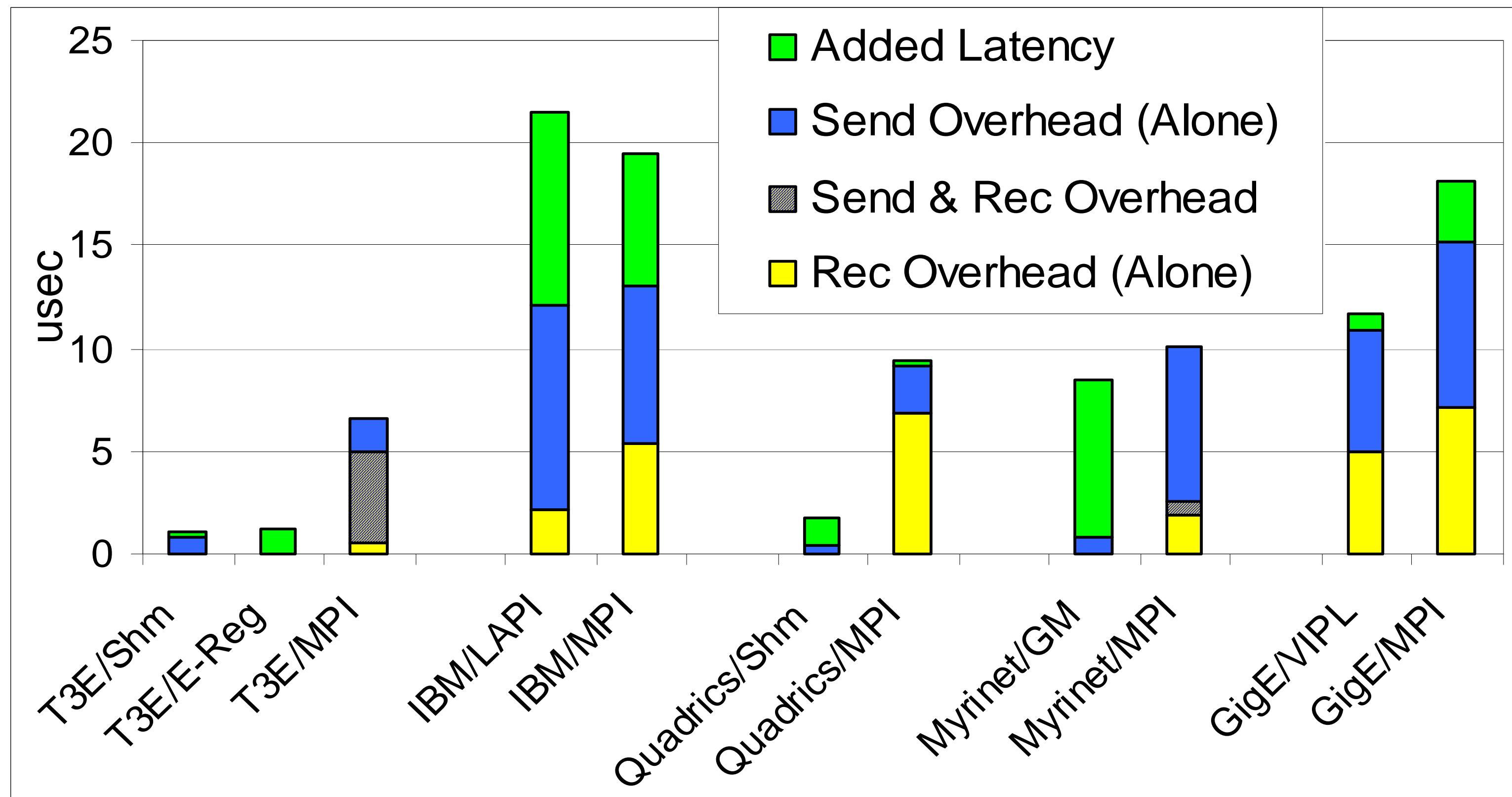
- Explicitly-parallel programming model with SPMD parallelism
  - Fixed at program start-up, typically 1 thread per processor
- Global address space model of memory
  - Allows programmer to directly represent distributed data structures
- Address space is logically partitioned
  - Local vs. remote memory (two-level hierarchy)
- Programmer control over performance critical decisions
  - Data layout and communication
- Performance transparency and tunability are goals
  - Initial implementation can use fine-grained shared memory
- Base languages differ: UPC (C), CAF (Fortran), Titanium (Java)

Global Address Space Eases Programming



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

One-Sided Communication May Improve Performance



- Potential performance advantage for fine-grained, one-sided programs
- Potential productivity advantage for irregular applications

Current Implementations

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

UPC Overview and Design Philosophy

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

UPC Execution Model

UPC Execution Model

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

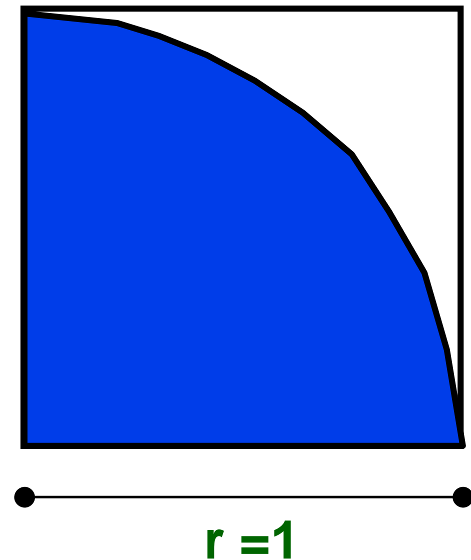
Hello World in UPC

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

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

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

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = ¼ * $\pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - π = 4*ratio



Pi in UPC

- Independent estimates of pi:

```
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;

    if (argc != 2)trials = 1000000;
    else trials = atoi(argv[1]);

    srand(MYTHREAD*17);

    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” separately

Helper Code for Pi in UPC

- Required includes:

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

- Function to throw dart and calculate where it hits:

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

Hidden slide

UPC Memory Model

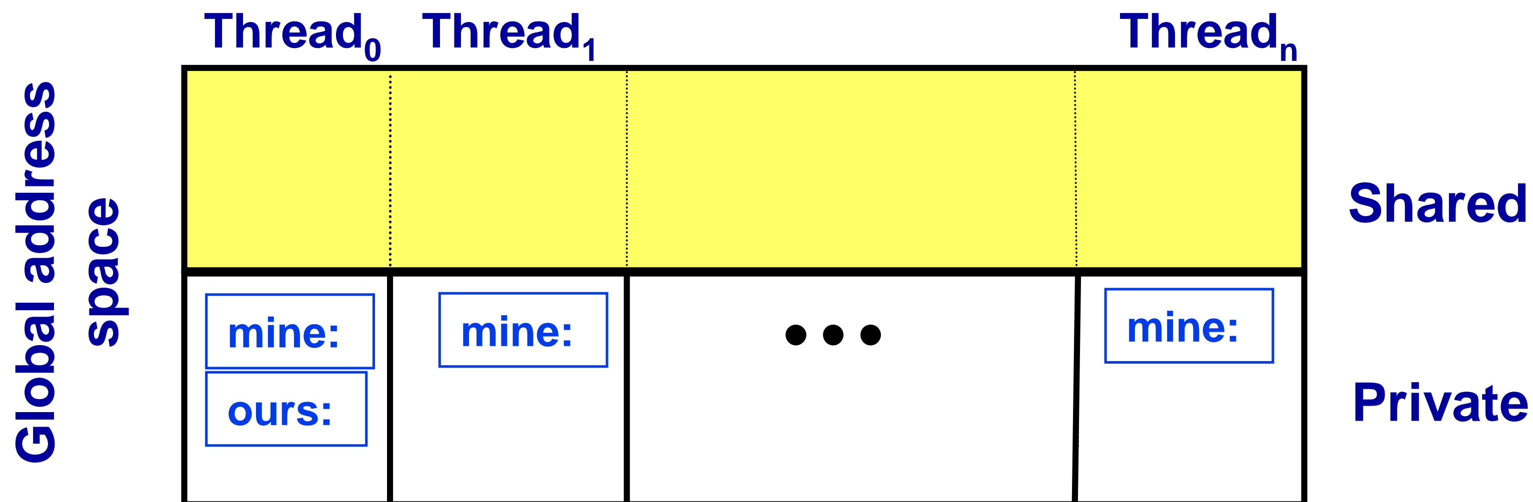
- Scalar Variables
- Distributed Arrays
- Pointers to shared data

Private vs. Shared Variables in UPC

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

```
shared int ours;
int mine;
```

- Simple shared variables of this kind may not occur in a within a function definition



Pi in UPC (Cooperative Version)

- Parallel computing of pi, but with a race condition

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

shared variable to record hits

divide work up evenly

accumulate hits

Pi in UPC (Cooperative Version)

- The race condition can be fixed in several ways:
  - Add a lock around the hits increment (later)
  - Have each thread update a separate counter:
    - **Have one thread compute sum**
    - **Use a "collective" to compute sum (recently added to UPC)**

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
  ... declarations an initialization code omitted
  for (i=0; i < my_trials; i++)
    all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++) hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
}
```

all_hits is shared by all processors, just as hits was

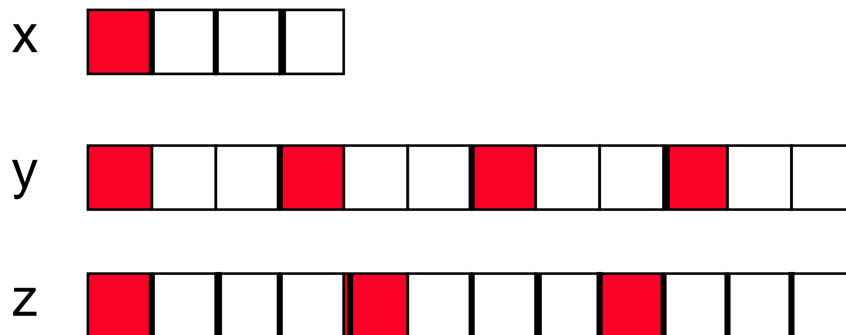
Where does it live?

Shared Arrays Are Cyclic By Default

- Shared array elements are spread across the threads

```
shared int x[THREADS]     /* 1 element per thread */
shared int y[3][THREADS]  /* 3 elements per thread */
shared int z[3*THREADS]   /* 3 elements per thread, cyclic */
```

- In the pictures below
  - Assume THREADS = 4
  - Elements with affinity to processor 0 are red



As a 2D array, this is logically blocked by columns

Example: Vector Addition

- Questions about parallel vector additions:
  - How to layout data (here it is cyclic)
  - Which processor does what (here it is “owner computes”)

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

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

Work Sharing with upc_forall()

- The idiom in the previous slide is very common
  - Loop over all; work on those owned by this proc
- UPC adds a special type of looIteration are independent

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

- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations to run
  - Integer: `affinity%THREADS` is `MYTHREAD`
  - Pointer: `upc_threadof(affinity)` is `MYTHREAD`
- Semantics are undefined if there are dependencies between iterations
  - Programmer has indicated iterations are independent

Vector Addition with upc_forall

- The vadd example can be rewritten as follows
  - Equivalent code could use "`&sum[i]`" for affinity
  - The code would be correct but slow if the affinity expression were `i+1` rather than `i`.

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

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

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

The cyclic data distribution may perform poorly on a cache-based shared memory machine

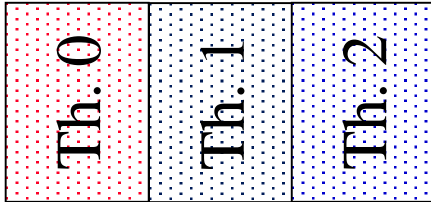
UPC Matrix Vector Multiplication Code

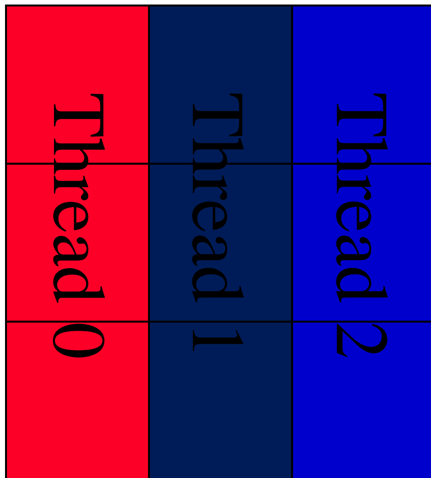
- The data distribution becomes more interesting in 2D
- Here is one possible matrix-vector multiplication

```
#include <upc_relaxed.h>

shared 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];
      }
}
```

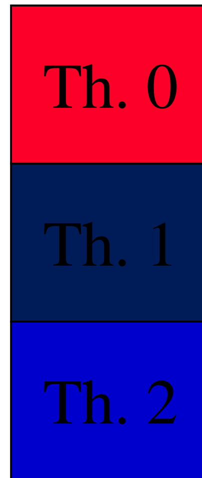
Data Distribution

B





*

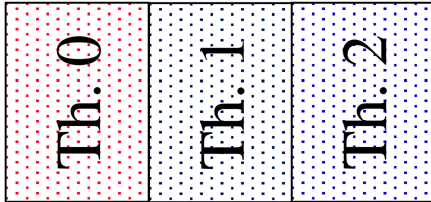


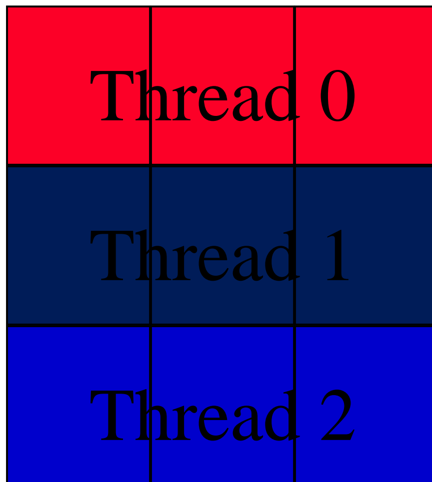
=

A B C

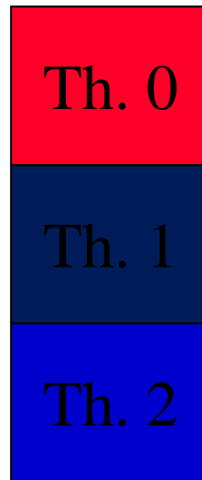
A Better Data Distribution

B





*



=

A B C

Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers.

  layout_specifier::

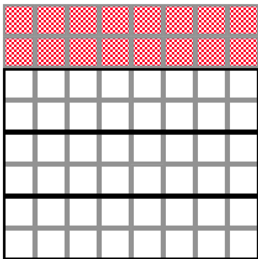
      null

      layout_specifier [integer_expression]
- The affinity of an array element is defined in terms of the
- block size, a compile-time constant, and THREADS a runtime constant.
- Element i has affinity with thread

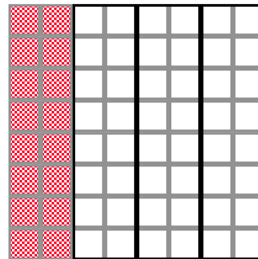
  (i / block_size) % PROCS.

Layout Terminology

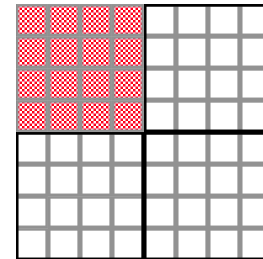
- Notation is HPF, but terminology is language-independent
  - Assume there are 4 processors



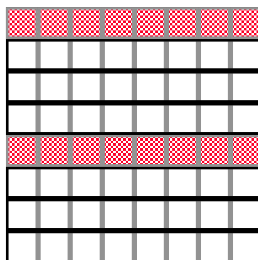
(Block, *)



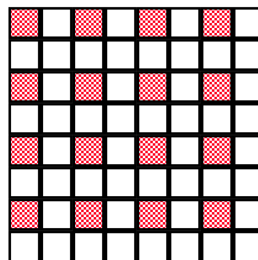
(*, Block)



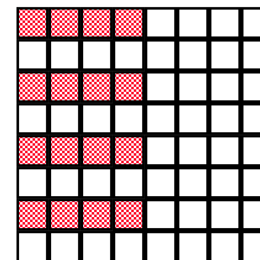
(Block, Block)



(Cyclic, *)



(Cyclic, Cyclic)



(Cyclic, Block)

2D Array Layouts in UPC

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

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

- If (k + m) % THREADS = = 0 them a3 has a row layout

  shared int a3 [n][m+k];

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

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

- or equivalently

  shared [b1*b2] int a5 [m][n][r][c][b1][b2];

UPC Matrix Vector Multiplication Code

- Matrix-vector multiplication with better layout

```
#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];
      }
}
```

Example: Matrix Multiplication in UPC

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

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

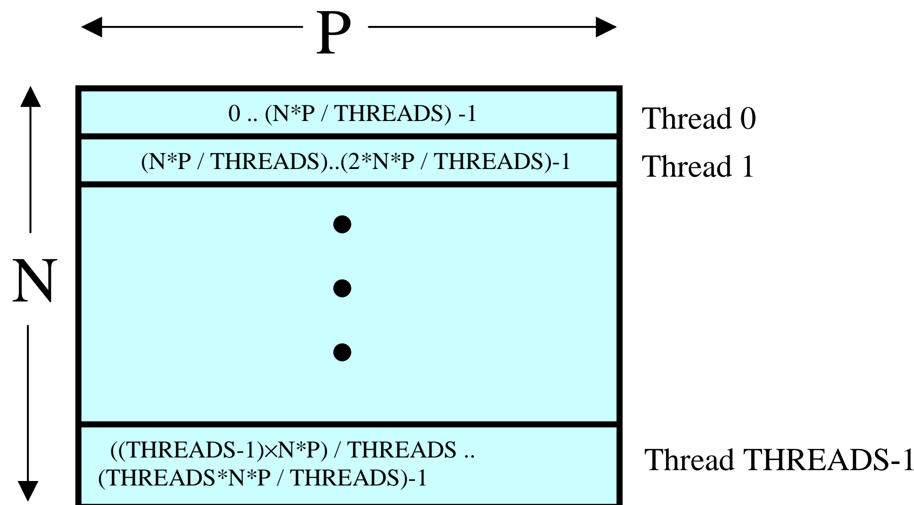
Matrix Multiply in C

```
#include <stdlib.h>
#include <time.h>

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

int a[N][P], c[N][M];
int b[P][M];

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];
   }
  }
}
```

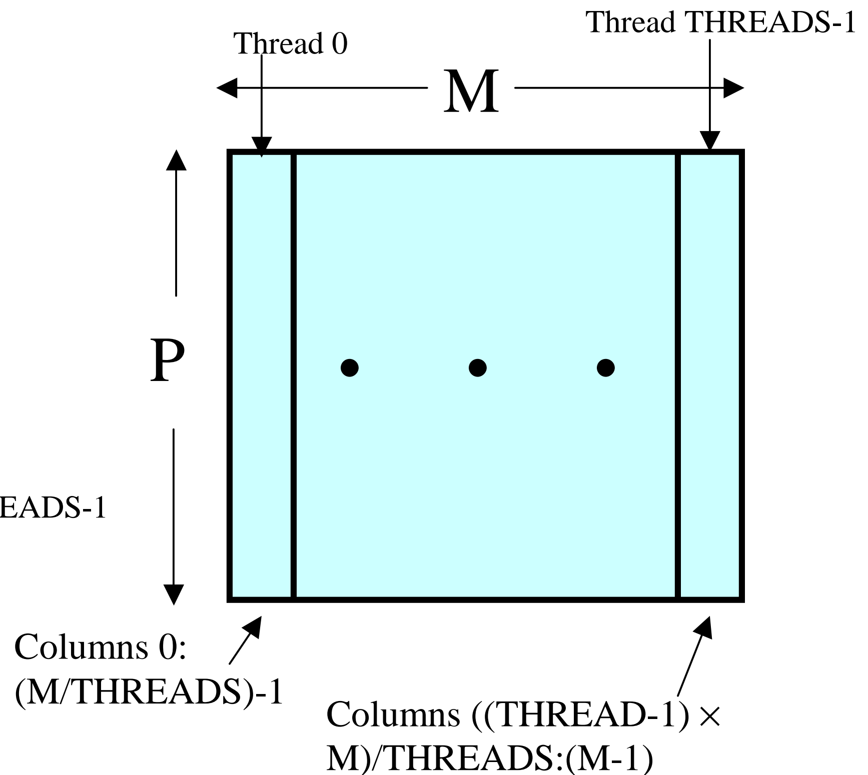
Domain Decomposition for UPC

- Exploits locality in matrix multiplication

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



- B($P \times M$) 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], c[N][M];
// a and c are row-wise blocked shared matrices

shared[M/THREADS] int b[P][M]; //column-wise blocking

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];
                }
        }
}
```

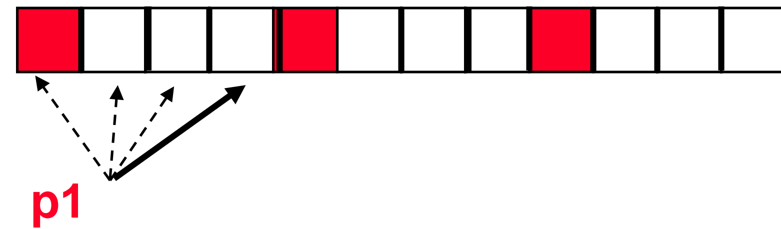
Notes on the Matrix Multiplication Example

- The UPC code for the matrix multiplication is almost the same size as the sequential code
- Shared variable declarations include the keyword shared
- Making a private copy of matrix B in each thread might result in better performance since many remote memory operations can be avoided
- Can be done with the help of upc_memget

Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
- Here is the vector addition example using pointers

```
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
   int i;
   shared int *p1, *p2;

   p1=v1; p2=v2;
   for (i=0; i<N; i++, p1++, p2++ )
      if (i %THREADS= = MYTHREAD)
            sum[i]= *p1 + *p2;
   }
```



UPC Pointers

Where does the pointer reside?

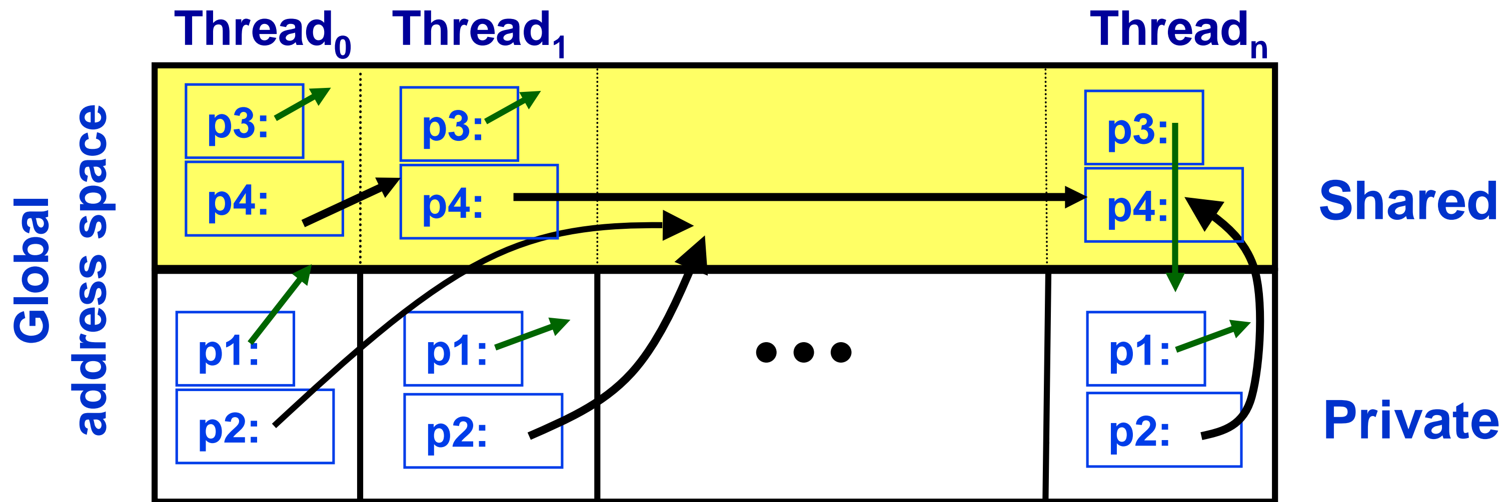
Where does it point?

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

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

Shared to private is not recommended.

UPC Pointers




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

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

Common Uses for UPC Pointer Types

```
int *p1;
```

- These pointers are fast
- Use to access private data in part of code performing local work
- Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

```
shared int *p2;
```

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication

```
int *shared p3;
```

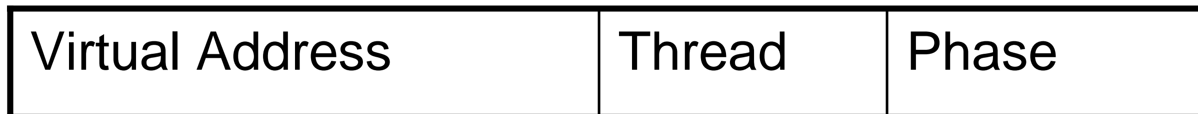
- Not recommended

```
shared int  *shared p4;
```

- Use to build shared linked structures, e.g., a linked list

UPC Pointers

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



- 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

Synchronization

- No implicit synchronization among the threads
- UPC provides many synchronization mechanisms:
  - Barriers (Blocking)
    - upc_barrier
  - Split Phase Barriers (Non Blocking)
    - upc_notify
    - upc_wait
  - Optional label allow for
  - Locks

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

Corrected version Pi Example

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

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

shared variable to record hits

accumulate hits

Memory Consistency in UPC

- The consistency model of shared memory accesses are controlled by designating accesses as strict, relaxed, or unualified (the default).
- There are several ways of designating the ordering type.
- A type qualifier, strict or relaxed can be used to affect all variables of that type.
- Labels strict or relaxed can be used to control the accesses within a statement.
- strict : { x = y ; z = y+1; }
- A strict or relaxed cast can be used to override the current label or type qualifier.

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

Matrix Multiplication with Blocked Matrices

```
#include <upc_relaxed.h>
shared [N*P/THREADS] int a[N][P], c[N][M];

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];
              }
       }
}
```

Shared and Private Data

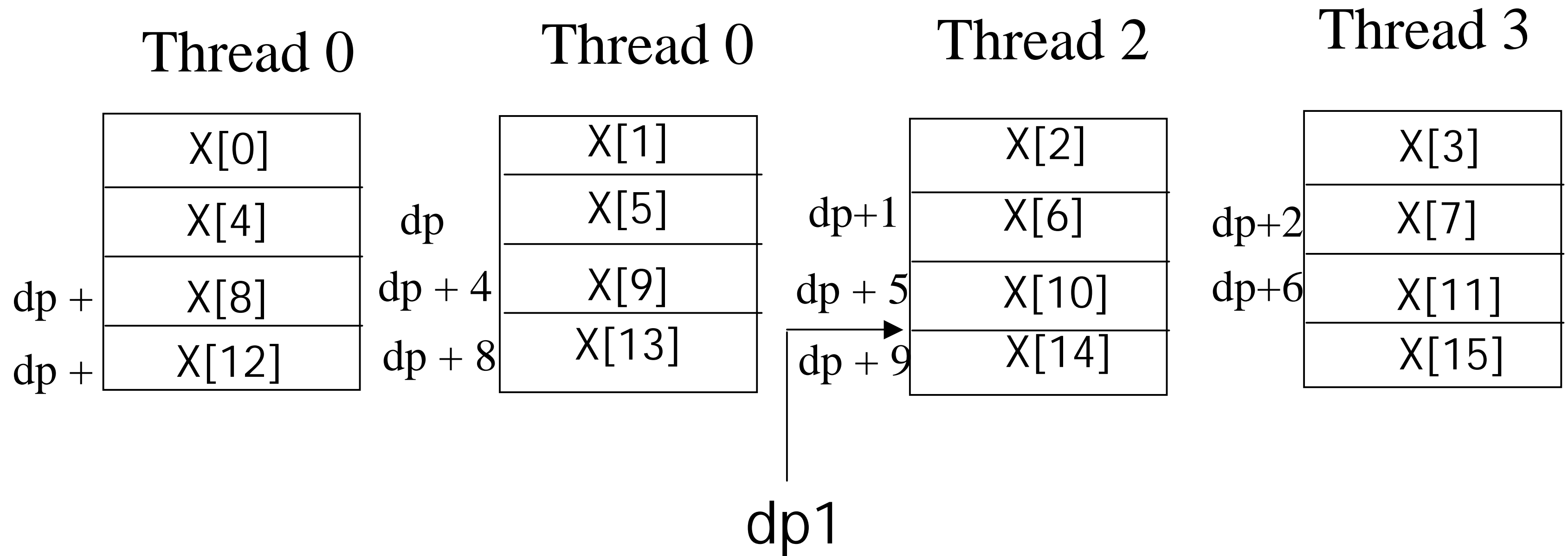
Assume THREADS = 4

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

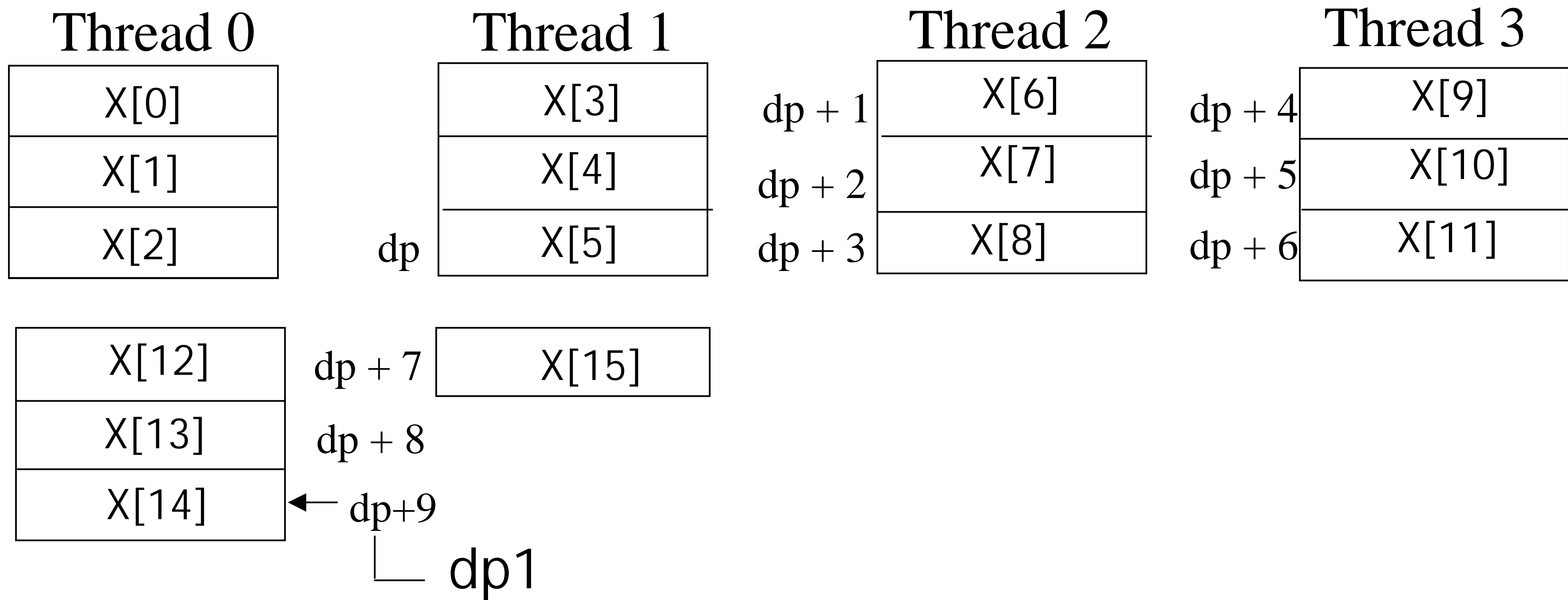
will result in the following data layout:

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



UPC Pointers



Bulk Copy Operations 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
- 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 happen 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

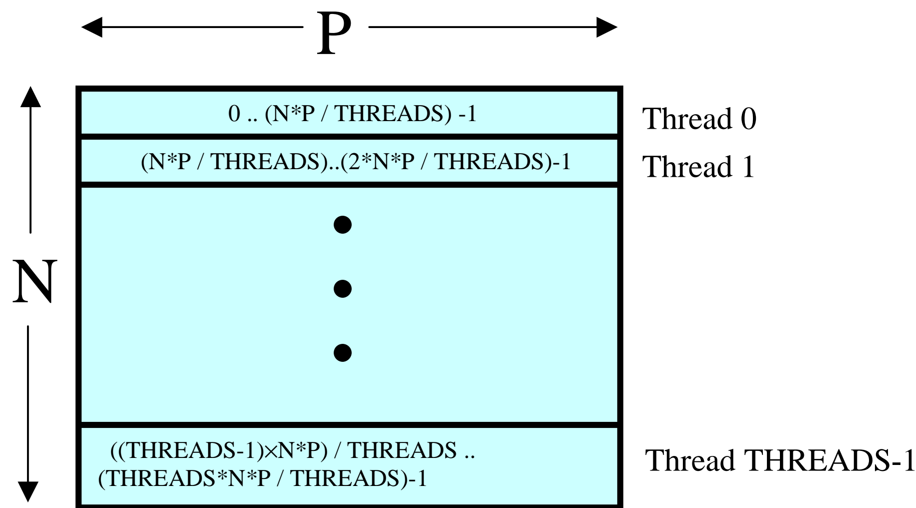
```
#include <stdlib.h>
#include <time.h>
#define N  4
#define P  4
#define M 4
int a[N][P] =  {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

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: some compiler do not yet support the intialization in declaration statements

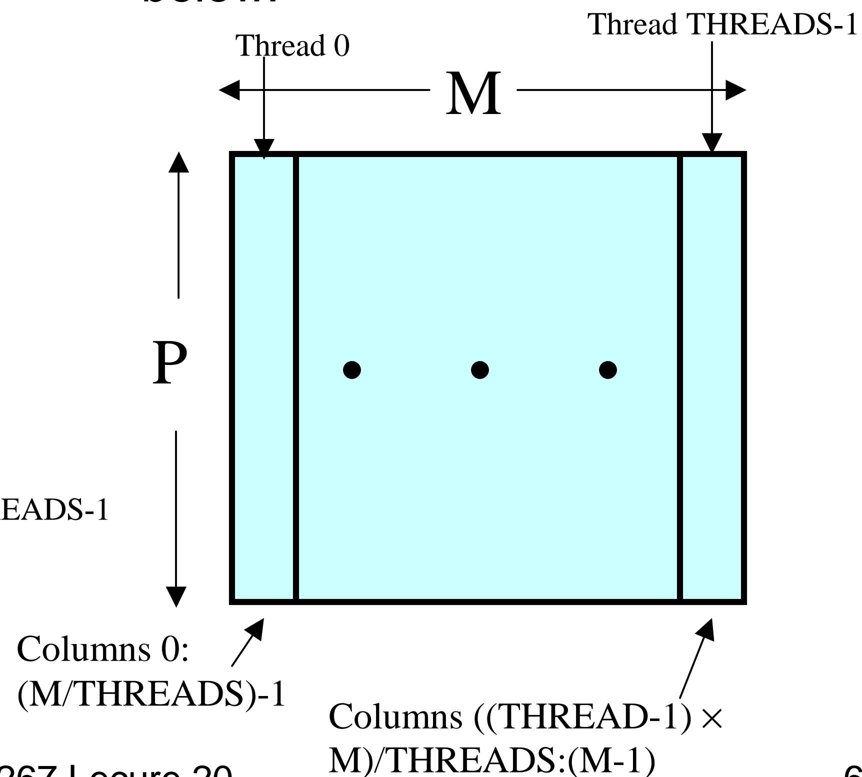
Domain Decomposition for UPC

- 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. <u>Synchronization</u>

9. Performance Tuning and Early Results

10. Concluding Remarks

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

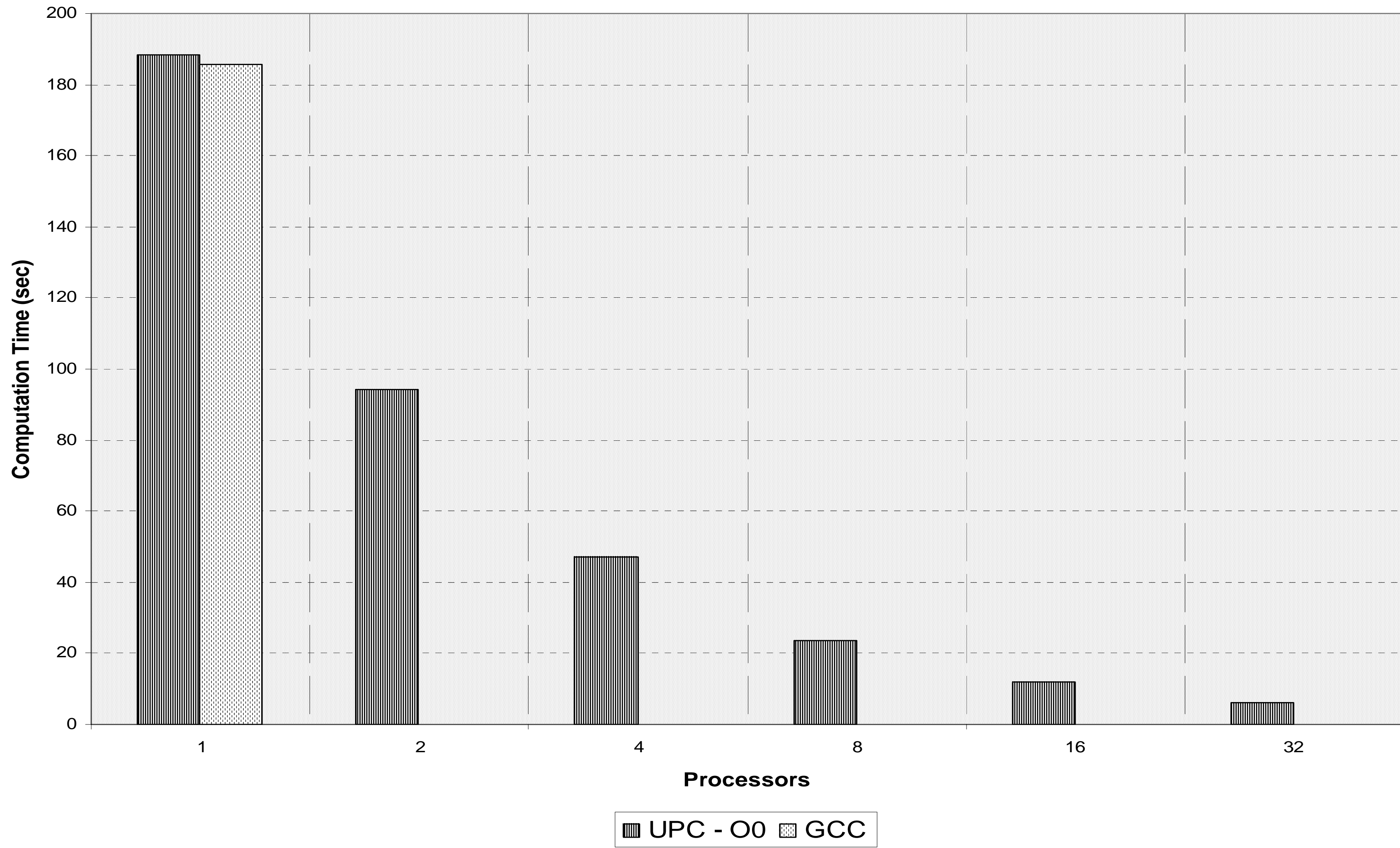
- UPC benchmarking results
  - Nqueens Problem
  - Matrix Multiplication
  - Sobel Edge detection
  - Stream and GUPS
  - NPB
  - Splash-2
- Compaq AlphaServer SC and Origin 2000/3000
- Check the web site for new measurements

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

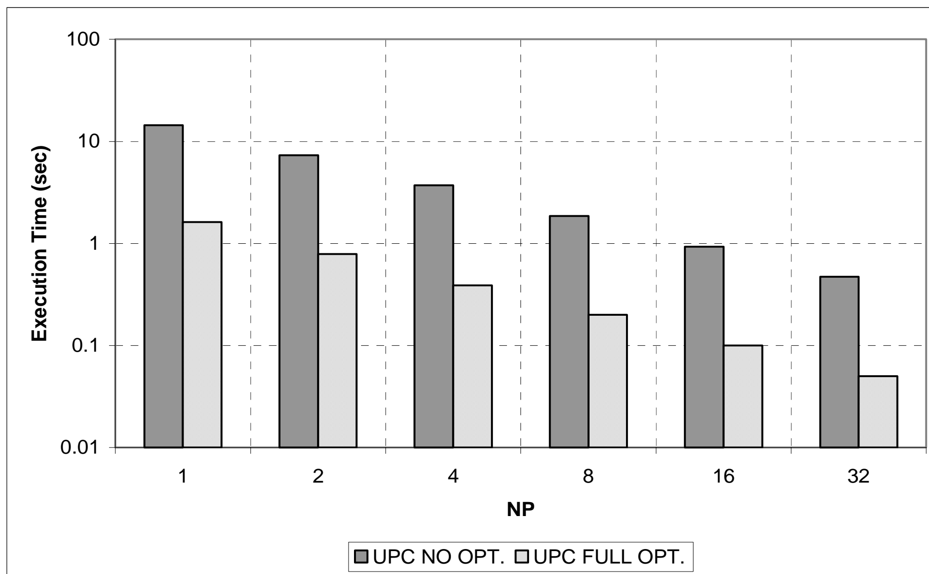
STREAM BENCHMARK

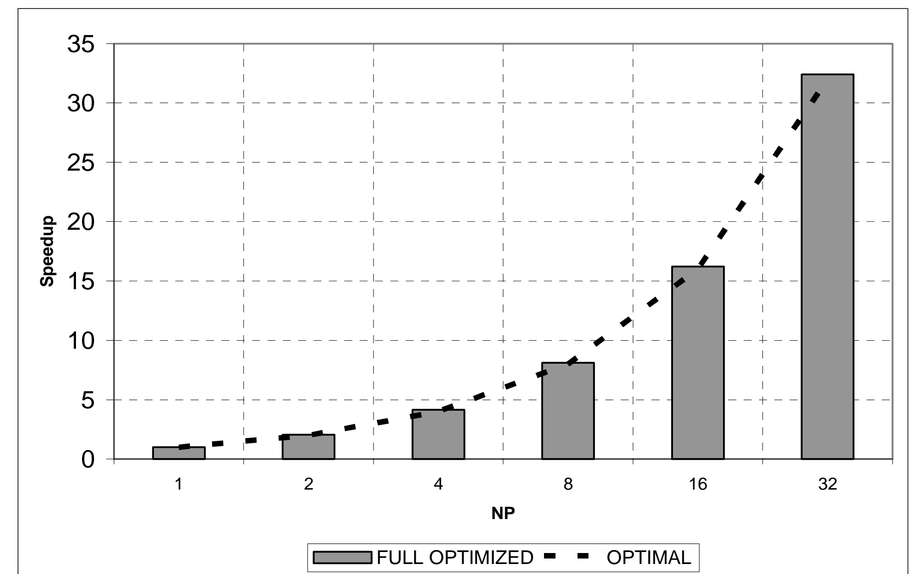
MB/S	Memcpy	Array Copy	Scale	Sum	Block Get	Block Scale
GCC	400	266	266	800	N/A	N/A
UPC Private	400	266	266	800	N/A	N/A
UPC Local	N/A	40	44	100	400	400
UPC Shared (SMP)	N/A	40	44	88	266	266
UPC Shared (Remote)	N/A	34	38	72	200	200

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



Performance of Edge detection on the Origin 2000

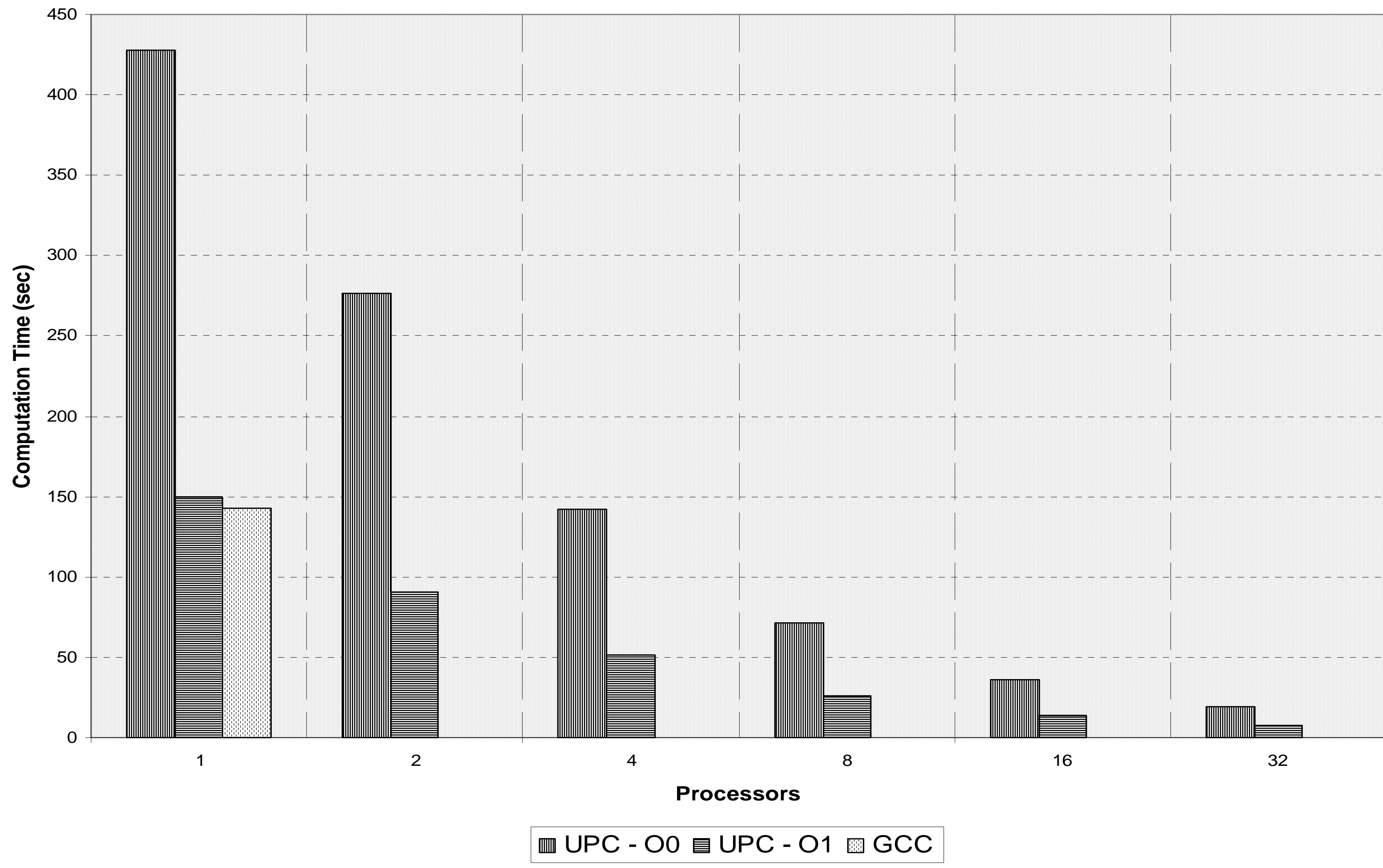




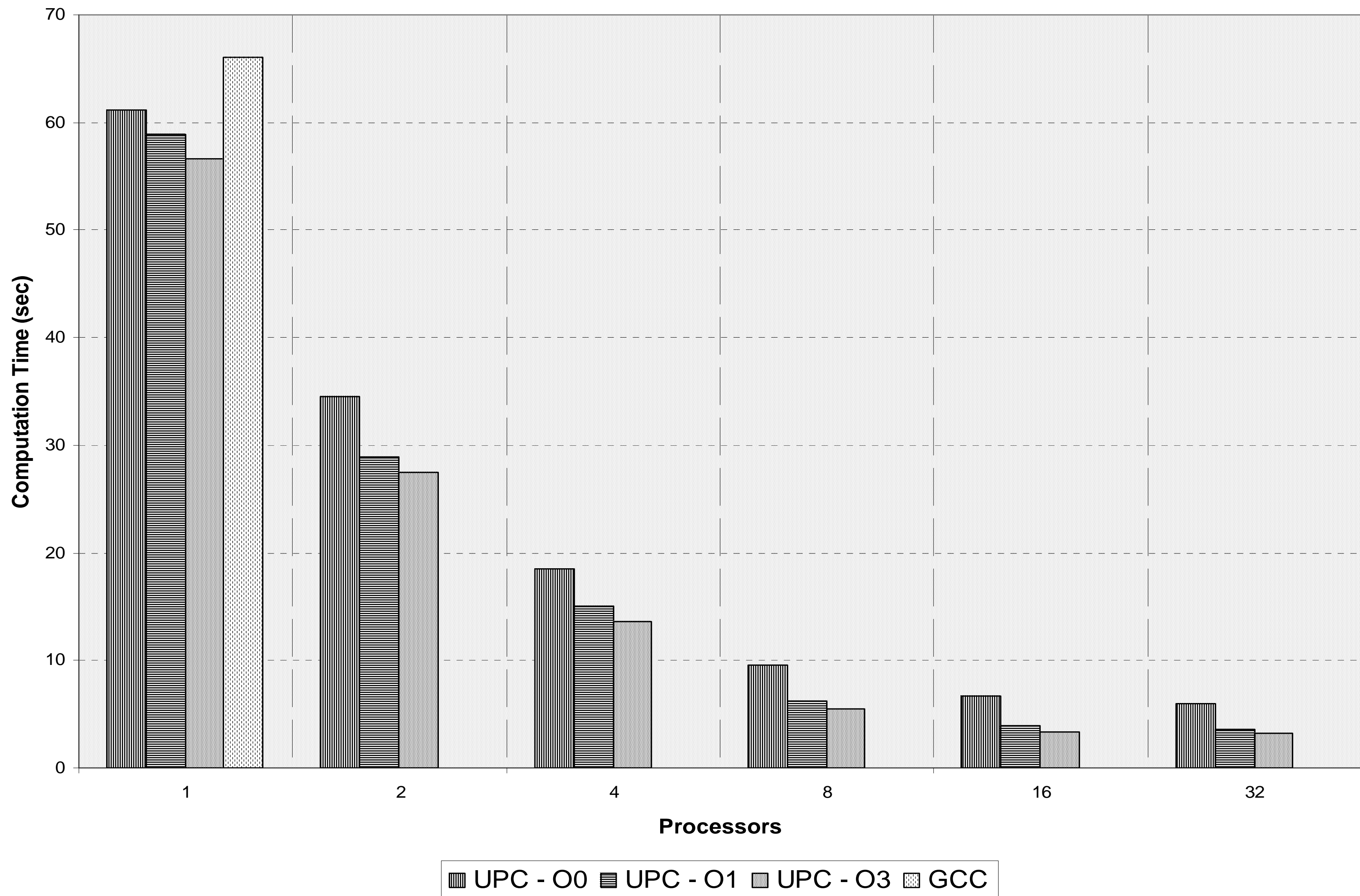
Execution Time

Speedup

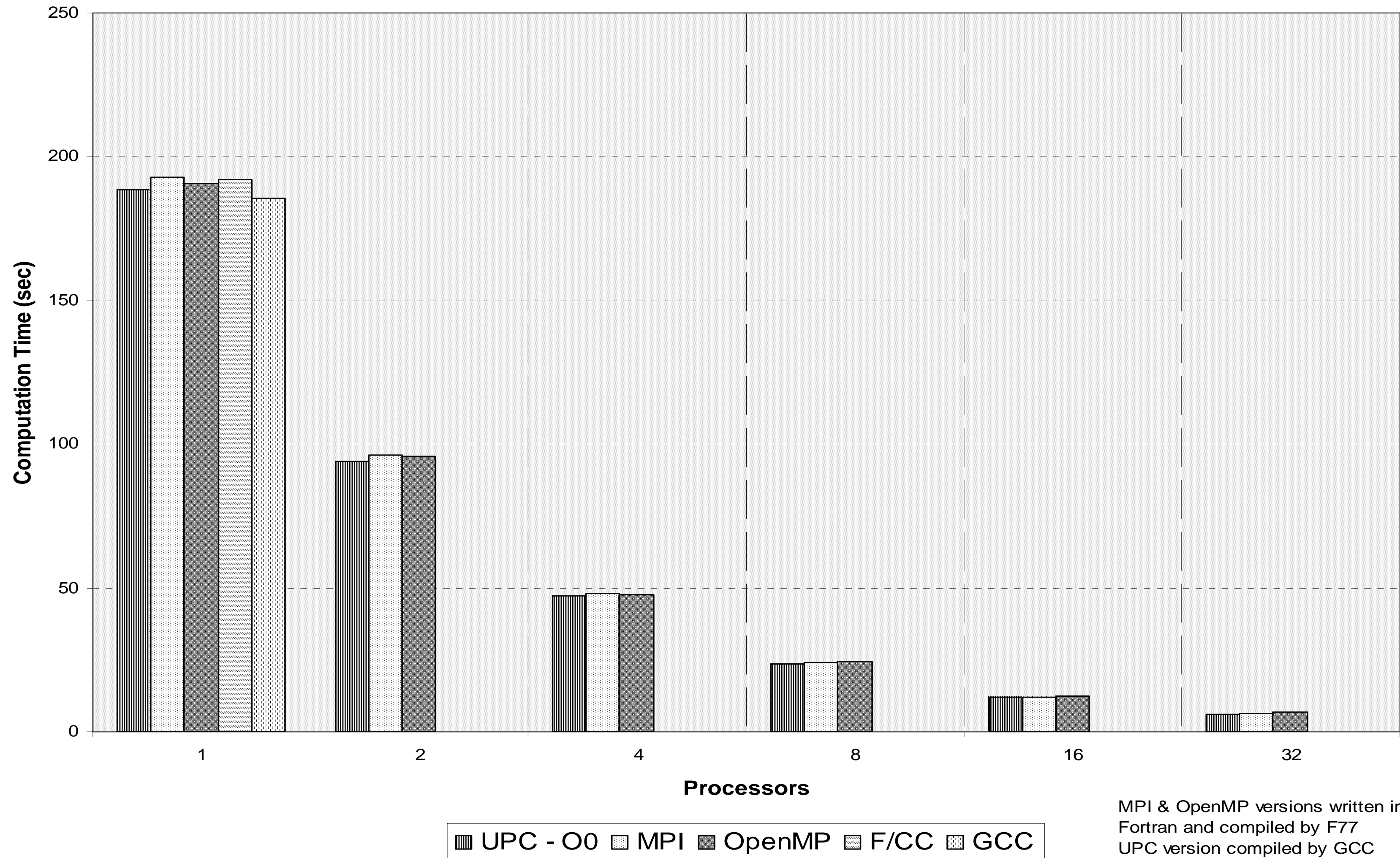
Execution Time over SGI–Origin 2k NAS-FT – Class A



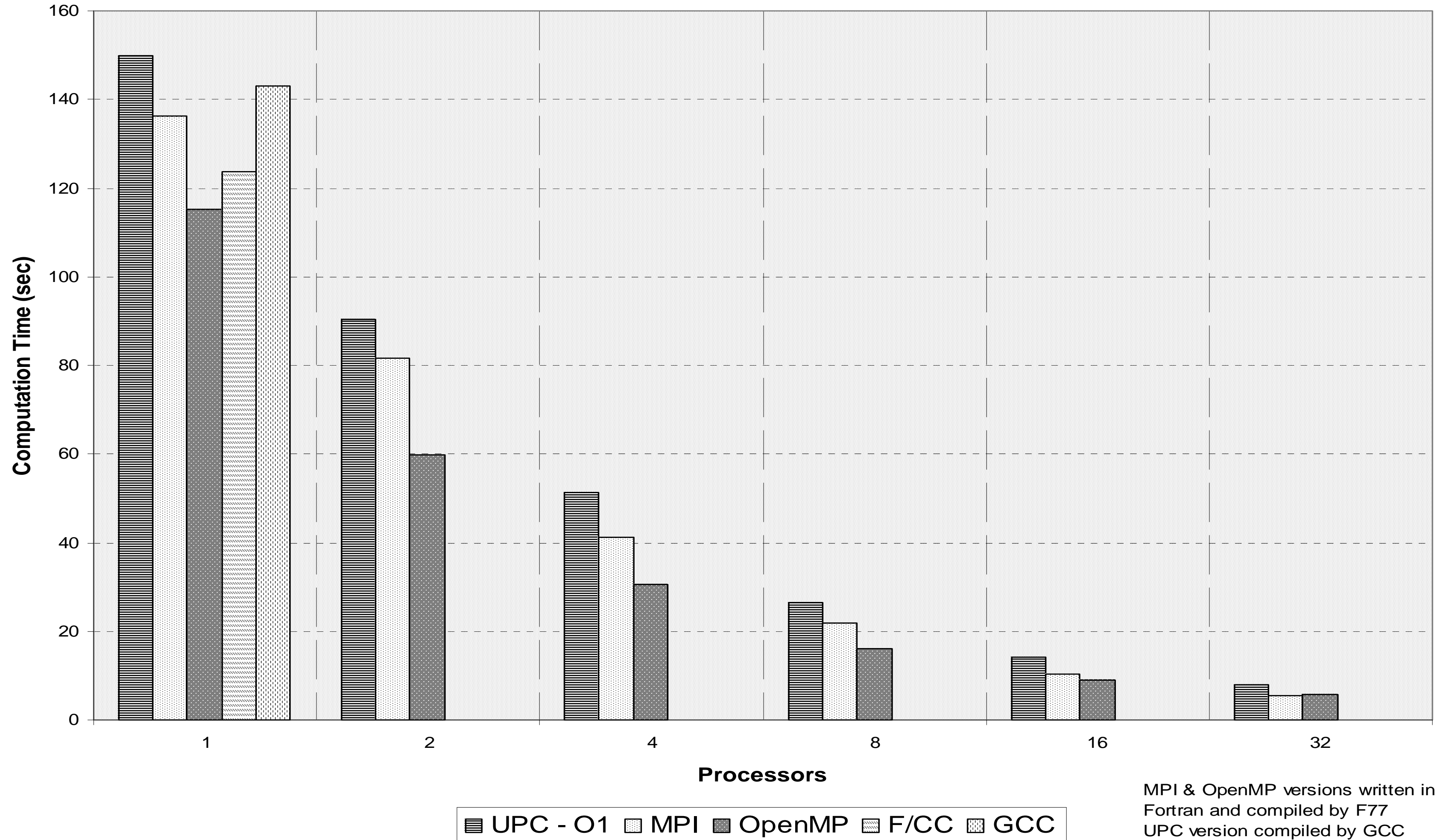
Execution Time over SGI–Origin 2k NAS-CG – Class A



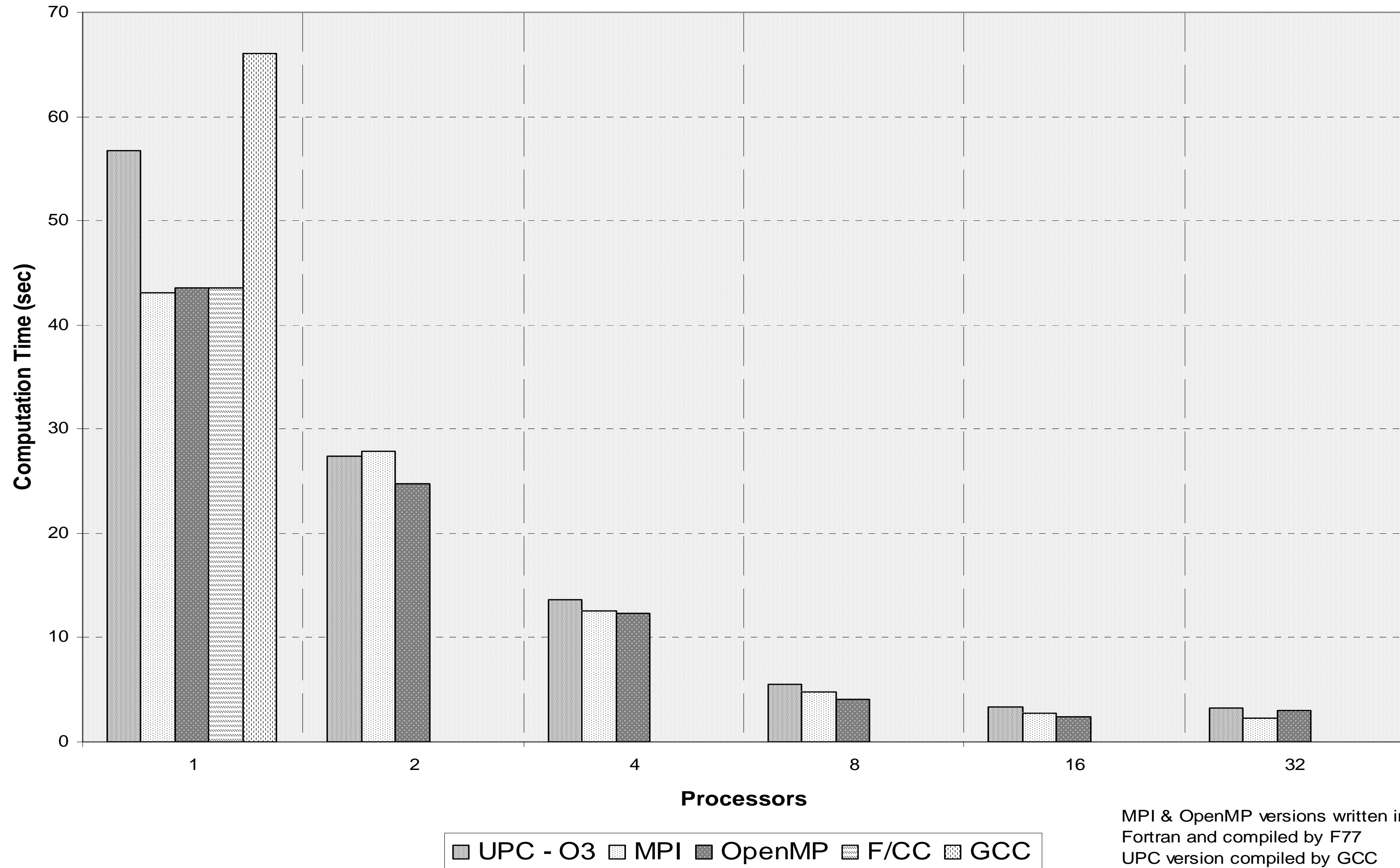
Execution Time over SGI–Origin 2k NAS-EP – Class A



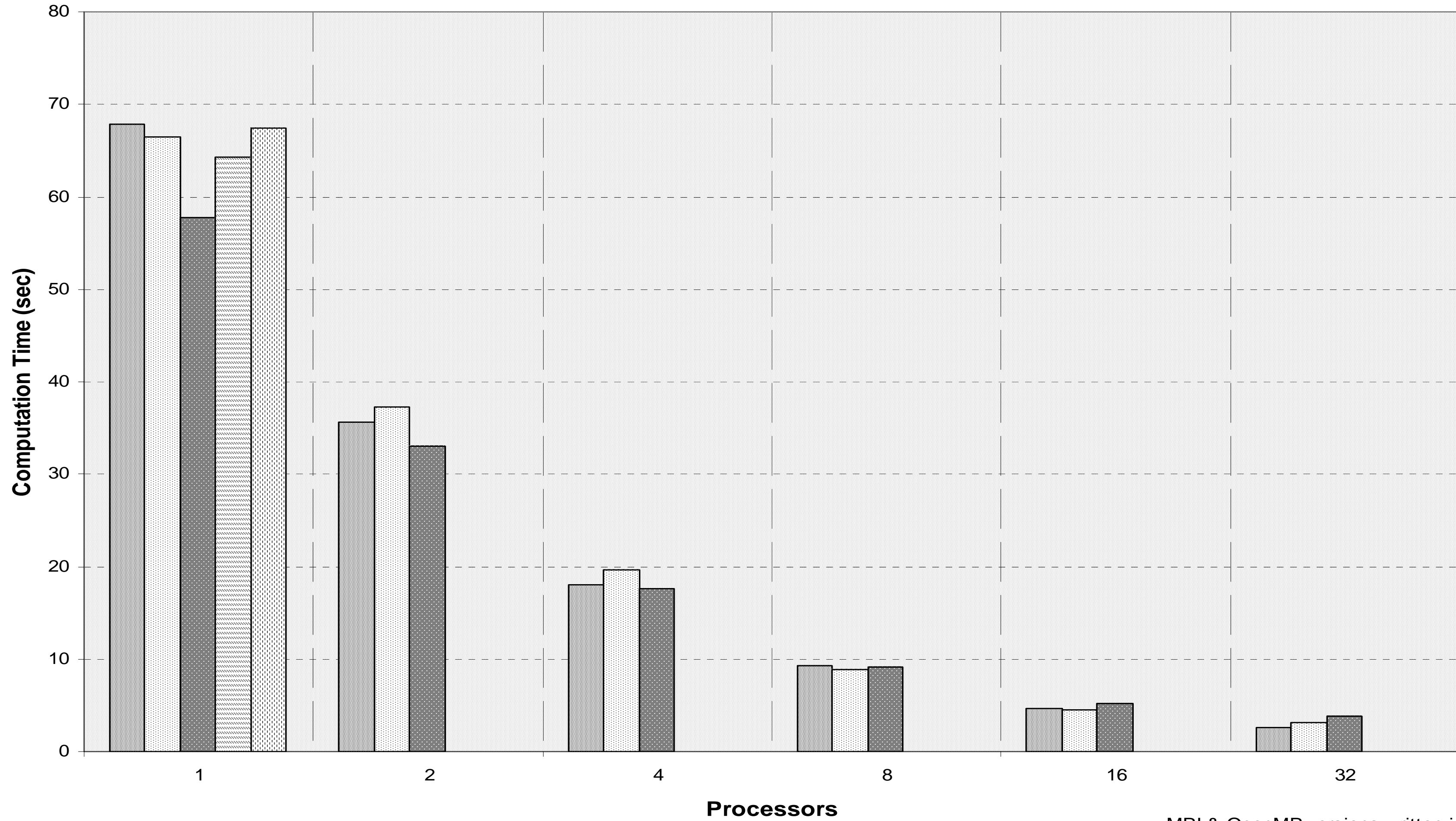
Execution Time over SGI–Origin 2k NAS-FT – Class A



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



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



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

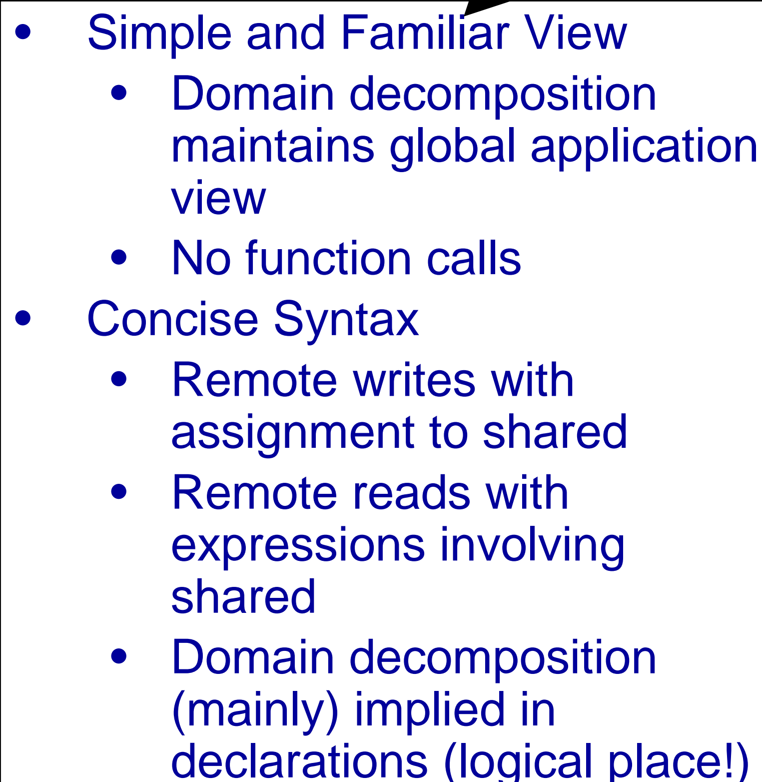
8. Synchronization

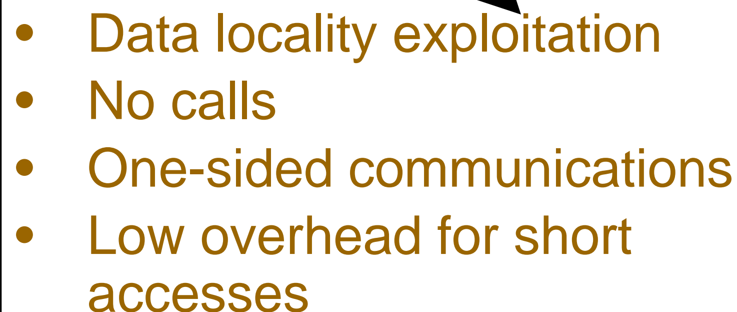
9. Performance Tuning and Early Results

10. Concluding Remarks

Conclusions

$UPC_{Time\text{-}To\text{-}Solution} = UPC_{Programming\ Time} + UPC_{Execution\ Time}$





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.

CS267 Final Projects

- Project proposal
  - Teams of 3 students, typically across departments
  - Interesting parallel application or system
  - Conference-quality paper
  - High performance is key:
    - Understanding performance, tuning, scaling, etc.
    - More important the difficulty of problem

- Leverage
  - Projects in other classes (but discuss with me first)
  - Research projects

Project Ideas

- Applications
  - Implement existing sequential or shared memory program on distributed memory
  - Investigate SMP trade-offs (using only MPI versus MPI and thread based parallelism)
- Tools and Systems
  - Effects of reordering on sparse matrix factoring and solves
- Numerical algorithms
  - Improved solver for immersed boundary method
  - Use of multiple vectors (blocked algorithms) in iterative solvers

Project Ideas

- Novel computational platforms
  - Exploiting hierarchy of SMP-clusters in benchmarks
  - Computing aggregate operations on ad hoc networks (Culler)
  - Push/explore limits of computing on “the grid”
  - Performance under failures
- Detailed benchmarking and performance analysis, including identification of optimization opportunities
  - Titanium
  - UPC
  - IBM SP (Blue Horizon)