Programming in UPC

Tarek El-Ghazawi

The George Washington University

tarek@seas.gwu.edu

Outline

Outline

The Short Story of UPC

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

$\Rightarrow$ **UPC !**

Design Philosophy

- Similar to the C language philosophy
  - Programmers are clever and careful
  - Programmers can get close to hardware
    - to get performance, but
    - can get in trouble
  - Concise and efficient syntax
- Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C

Design Philosophy

Design Philosophy

- Allow easy implementations onto different architectures
- Provide high-performance at the
  - Node level
  - System Level

Design Philosophy

Outline

Outline

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 GMU, IDA, NSA and GMU.
- The participants currently are: ARSC, Compaq, CSC, Cray Inc., Etnus, GMU, HP, IBM, IDA CSC, Intrepid Technologies, LBNL, LLNL, MTU, NSA, SGI, Sun Microsystems, UCB, US DOD

Status

History

- First consortium meeting, May 18-19 of 2000, in Bowie, Maryland.
- The second meeting has been held in November 2000, in Dallas.

Status

Outline

- Design Philosophy
- History
- Status
- Background
  - Parallel programming models
  - What is UPC ?
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

Status

- Specification v1.0 completed February of 2001
- Benchmark, UPC_Bench, v1.0pre1, released by GMU
- Testing suite released by GMU, v1.0pre1.
- Course continuously updated and offered at GMU, NSA, and in the UK.
- Research Exhibit at Supercomputing 2000, Dallas.
- UPC web site

Status

UPC Web Site

http://upc.gwu.edu

Status

Hardware Platforms

- UPC implementations are available for
  - Cray T3D/E (V3.1.9)
  - Compaq AlphaServer SC (V1.51)
- Ongoing and future implementations for:
  - HP
  - Sun multiprocessors
  - Cray SV-2
  - SGI
  - IBM
  - Possibly Beowulf Clusters

Outline

- Design Philosophy
- History
- Status
- Background
  - Parallel programming models
  - What is UPC ?
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

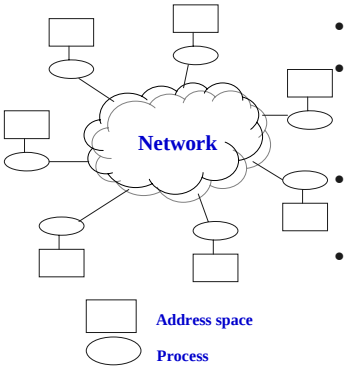
Outline

Parallel Programming Paradigms/Models

- Programmers view of data and execution under the used programming method
- Should not be confused with parallel architectures such as SIMD, MIMD, …, although some similarities exist
- Simplify programming independent of underlying architecture
- Simplify efficient mapping of computations onto actual architectures

Background

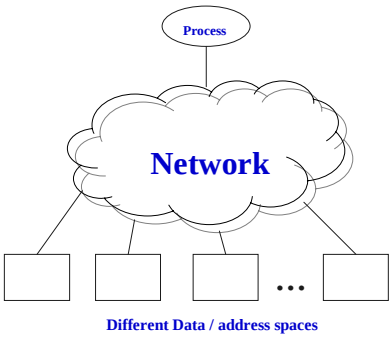
The Message Passing Model



- A set of cooperating sequential processes
- Each has a local address space
- Processes interact via explicit communication transactions (sends, receives, …)
- (+)ive: Programmers control data and work distribution.
- (-)ive:
  - Significant communication overhead for small transactions
  - Hard to program in
- Example: MPI

Background

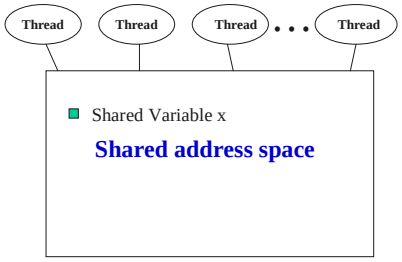
The Data Parallel Model



- One thread (process) of execution
- Different data items are manipulated in the same way by that thread
- Conditional statements to exclude (or include) part(s) of data in an operation
- (+)ive: Easy to write and comprehend, no synchronization required
- (-)ive: No independent branching
- Example: HPF

Background

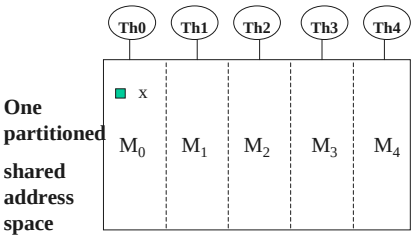
The Shared Memory Model



- Different simultaneous execution threads (precesses)
- One shared address space
- (+)ive:
  - Simple statements
    - read remote memory via an expression
    - write remote memory through assignment
- (-)ive:
  - Manipulating shared data leads to synchronization requirements
  - Does not allow locality exploitation
- Example: OpenMP

Background

The Distributed Shared Memory Model

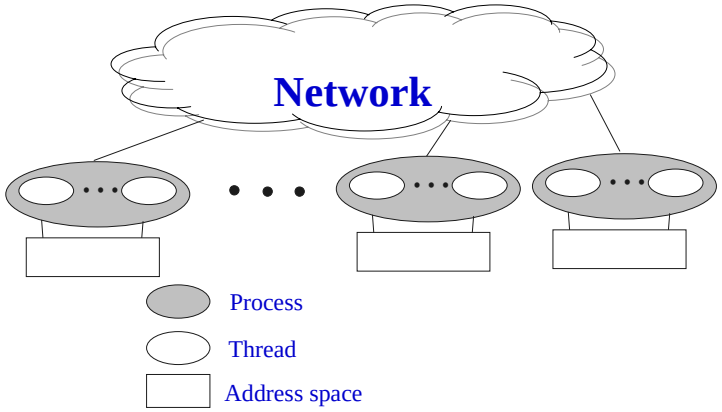


- Similar to the shared memory paradigm
- Memory M_i has affinity to thread Th_i
- (+)ive:
  - Helps exploiting locality of references
  - Simple statements as SM
- (-)ive: Synchronization
- Example : UPC

Background

Hybrid Model(s)

Example: Shared + Message Passing



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

Background

Outline

- Design Philosophy
- History
- Status
- Background
  - Parallel programming models
  - What is UPC ?
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

What is UPC?

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

Background

UPC is Portable

- Establishes a common language syntax and semantics for parallel programming in C
- A simple parallel extension of the standard ANSI C

Background

UPC is Easy to Use

- Reads remote memory with expression syntax
- Writes remote memory with assignment statement
- No need to worry about the native programming model or hardware architecture
- Local and remote data are differentiated solely by their declarations

Background

UPC is Efficient

- Provides the programmer with a simple but effective access/view to the underlying machine
- Provides a direct and easy mapping from language to machine instructions
- Allows minimization of thread communication overhead and achieves high performance

Background

Outline

- Design Philosophy
- History
- Status
- Background
  - Parallel programming models
  - What is UPC ?
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

UPC Highlights

- Simple DSM extensions to ANSI C
  - New keywords for shared data
  - Blocking of shared arrays for easy domain decomposition
  - Control for **affinity** of data and threads, for better locality exploitation

Programming Model

UPC Highlights

- Simple memory model
  - Private
  - Shared
- Simple execution model
  - One thread per processor (on current implementations, but more are possible)
  - Straightforward synchronization primitives
- High performance Implementation
  - C performance for local operations
  - Small overhead for remote operations

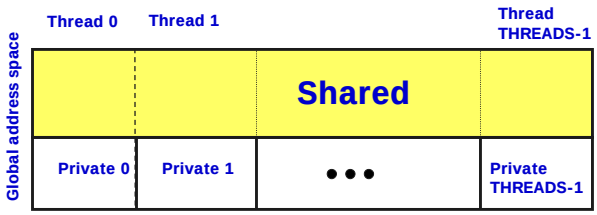
Programming Model

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 private space and a portion of the shared address space.

Programming Model

Memory View



•A shared pointer can reference all locations in the shared space
•A private pointer may reference only addresses in its private space or addresses in its portion of the shared space
•Static and dynamic memory allocations are supported for both shared and private memory

Parallelism/Thread View

- A number of threads working independently
- MYTHREAD specifies thread index (0..THREADS-1)
- Simple synchronization
  - Barriers (upc_barrier())
  - Split Barriers (non-blocking)
    - upc_notify()
    - upc_wait()
  - Locks

Outline

- Design Philosophy
- Background
- Status
- UPC Programming Model
- Introduction to UPC
  - Vector addition examples
  - Compiling and running
  - Data declaration
  - UPC Pointers
  - Synchronizatio
  - Worksharing
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. More Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Conclusion

Outline

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++; i)
          If (MYTHREAD==i%THREADS)
               v1plusv2[i]=v1[i]+v2[i];
}
```

Introduction to UPC

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];
}
```


Notes on the example

- Use of the shared type qualifier for v1 v2 and v1plusv2
- v1, v2, and v1plusv2 are:
  - shared qualified
  - Distributed across threads, 100 elements per threads
  - Accessible by all threads
- upc_forall is a distributed for loop
- The 4th field of the upc_forall distributes the iterations, in this case one per thread in a round robin fashion

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

Matrix Multiplication

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 (on thread 0) */z

UPC Pointers

- Pointer declaration:

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

Shared Pointers:A Third Example

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

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

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

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


Synchronization - Barriers

- No implicit synchronization among the threads
- UPC provides the following barrier synchronization constructs:
  - Barriers (Blocking)
    - upc_barrier;

Barrier example

```
#include <upc_relaxed.h>

shared int result;

void main()
{
    int local_a, local_b;

    if (MYTHREAD==0)
        result = computation();
    upc_barrier;                    // Synchronization
    local_b=result*local_a;
}
```


Work Sharing

- Doing it explicitly:

```
for(j=0;j<THREADS*10;j++)
    if (MYTHREAD==j%THREADS)
    {
        … iteration body …
    }
```

- Each thread will execute 10 iterations
- Overhead of one additional test per iteration

Work Sharing with upc_forall()

- upc_forall construct is useful for SPMD programming
- Allows specification of work sharing
- Iterations are independent and executed in any order desired by the compiler and the runtime system

Work Sharing with upc_forall()

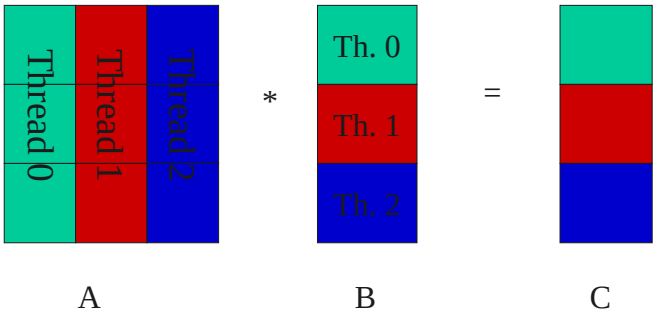
- Each thread gets a bunch of iterations.
- Affinity field to distribute work
- Simple C-like syntax and semantics

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

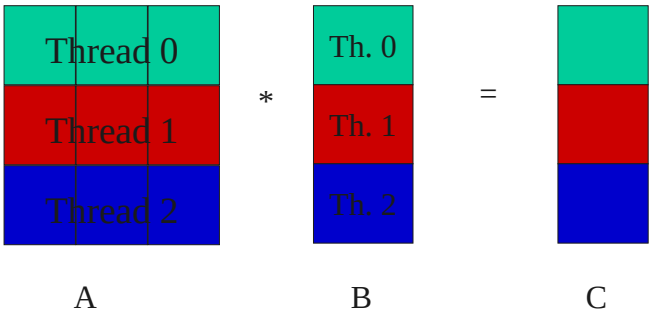
UPC Vector Matrix Multiplication Code

```
// vect_mat_mult.c

#include <upc_relaxed.h>

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

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

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


Data Distribution



A Better Data Distribution



Outline

Shared and Private Data

Examples of Shared and Private Data Layout:

Assume THREADS = 3

shared int x; /*x will be aligned with thread 0 */

shared int y[THREADS];

int z;

will result in the layout:

Thread 0	Thread 1	Thread 2
y[0]	y[1]	y[2]
x		
z	z	z

Data and Pointers

Shared and Private Data

shared int A[4][THREADS];

will result in the following data layout:

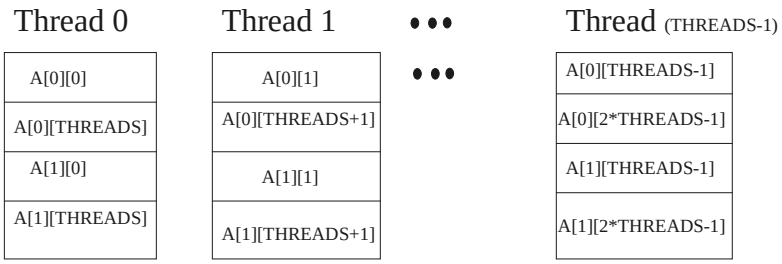
Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

Data and Pointers

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

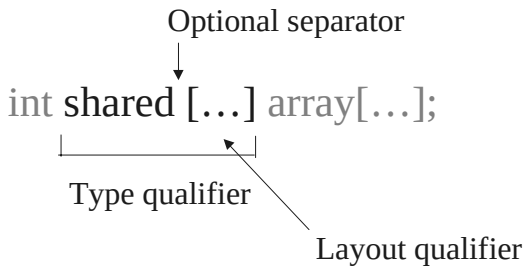
Assume THREADS = 4

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

will result into the following data layout:

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

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



Spacing Does Not Matter!

Examples:

- int shared [5] int a[10];
- int shared[5] int a [10];
- int shared [5] int a [10];
- int shared[5] int a[10];

Outline

Outline

UPC Pointers

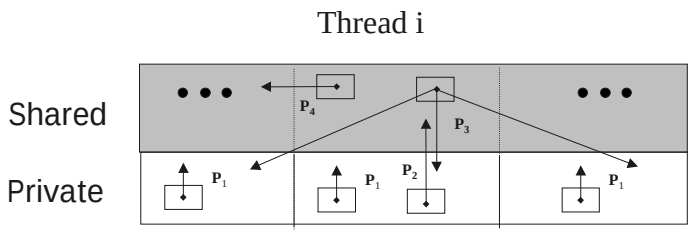
Where does the pointer reside?

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

UPC Pointers

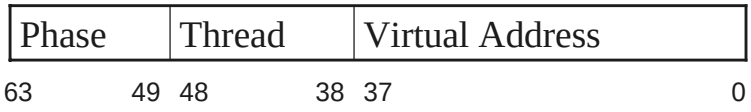


UPC Pointers

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

UPC Pointers

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



UPC Pointers

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa !
- When casting a shared pointer to a private pointer, the thread number of the shared pointer may be lost

Special Functions

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

Special Operators

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

Usage Example of Special Operators

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

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

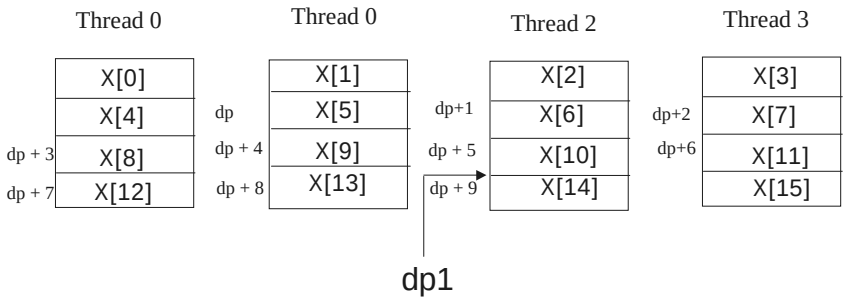
UPC Pointers

Shared Pointer Arithmetic Examples:

Assume THREADS = 4

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

UPC Pointers

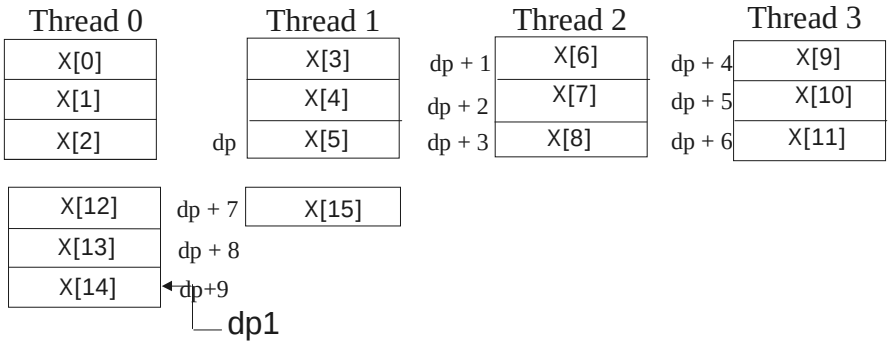


UPC Pointers

Assume THREADS = 4

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

UPC Pointers



Data and Pointers

UPC Pointers

Example Pointer Castings and Mismatched Assignments:

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

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

Data and Pointers

UPC Pointers

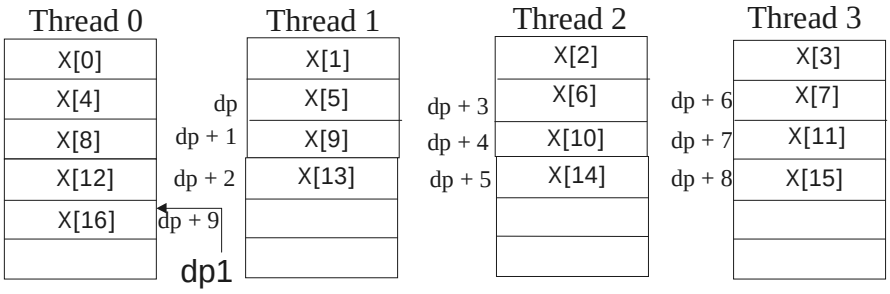
Assume THREADS = 4

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

- This statement assigns to dp1 a value that is 9 positions beyond dp
- The pointer will follow its own blocking and not the one of the array

Data and Pointers

UPC Pointers



Data and Pointers

UPC Pointers

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

Shared Pointers and Indirections

```
double *shared ptr;      //ptr is in shared
                         space
double *shared *ptr;     //*private to
                         shared to double
double *shared **ptr; //**private to
            private to shared to double
```

More Shared pointer Examples

- shared int *p2;
- shared int **p2;
- shared int *shared p2;
- shared int *shared *shared p2;
- shared int **shared p2;
- shared int *shared *p2;
- int *shared *p2;

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];
```

Data and Pointers

- 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..99	0
25..49	100..199	1
50..74	200..299	2
75..99	300..399	3

Outline

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

Outline

- Design Philosophy
- Background
- Status
- UPC Programming Model
- Introduction to UPC
- Data and pointers
  - Blocking
  - Advanced pointer concepts
  - Advanced forall
  - String functions
  - Example: Matrix multiplication
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. 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:

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

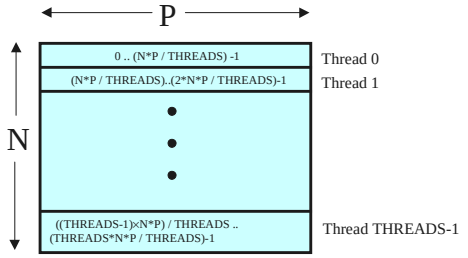
Doing it in C

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

03 #define N  4
04 #define P  4
05 #define M 4

06 int a[N][P] =  {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
07 int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

08 void main (void) {
09    int i, j , l;
10    for (i = 0 ; i<N ; i++) {
11          for (j=0 ; j<M ;j++) {
12                c[i][j] = 0;
13                for (l = 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
14          }
15    }
16 }
```

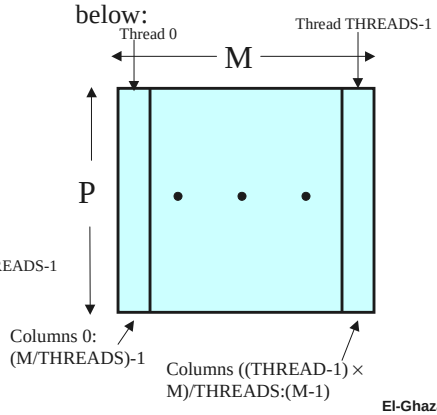

Domain Decomposition for UPC

- Exploits locality in matrix multiplication

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



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



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

UPC Matrix Multiplication Code

```
// mat_mult_1.c

#include <upc_relaxed.h>

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

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

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

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


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

Outline

- Design Philosophy
- Background
- Status
- UPC Programming Model
- Introduction to UPC
- Data and pointers
- Dynamic memory allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

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

Memory Allocation

Global Memory Allocation

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

nblocks : number of blocks
nbytes : block size

- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory space in the shared space
- If called by more than one thread, multiple regions are allocated and each thread, which makes the call, gets a different pointer
- Space allocated per calling thread is equivalent to : **shared [nbytes] char[nblocks * nbytes]**
- (Not yet implemented on Cray)

Collective Global Memory Allocation

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

nblocks: number of blocks
nbytes: block size

- This function has the same result as **upc_global_alloc**. But this is a collective function, which is expected to be called by all threads
- All the threads will get the same pointer
- Equivalent to : **shared [nbytes] char[nblocks * nbytes]**

Local Memory Allocation

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

nblocks : number of blocks
nbytes : block size

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

Memory Freeing

void upc_free(shared void *ptr);

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

Matrix Multiplication with dynamic memory

```
// mat_mult_2.c

#include <upc_relaxed.h>

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

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

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

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


Outline

- Design Philosophy
- Background
- Status
- UPC Programming Model
- Introduction to UPC
- More data and pointers
- Dynamic memory allocation
- Programming examples

9. More synchronization
10. Tuning
11. Performance results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Conclusion

Example: Sobel Edge Detection

Original Image

Edge-detected Image

Programming example

Sobel Edge Detection

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

Programming example

Template Convolution

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

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

Template

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

Image

Programming example

Sobel Edge Detection Masks

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

West Mask

Used to detect vertical edge

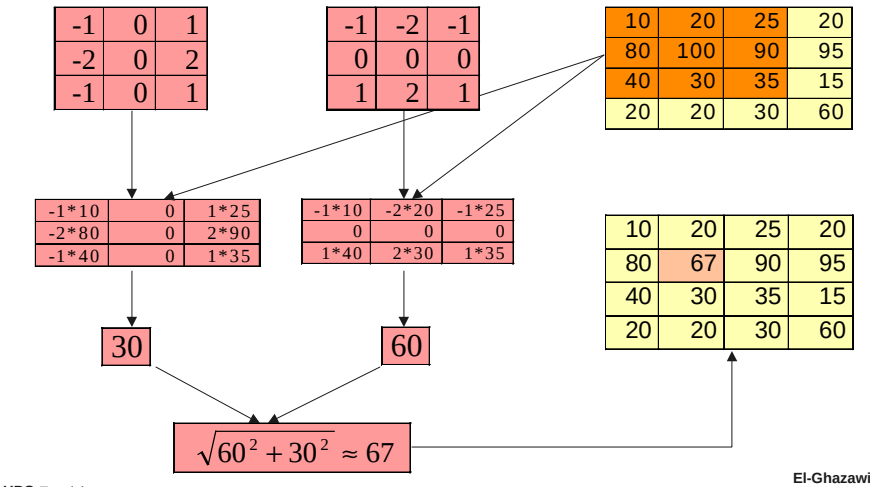
-1	-2	-1
0	0	0
1	2	1

North Mask

Used to detect horizontal edge

Programming example

Applying the Masks to an image



Programming example

Sobel Edge Detection – The C program

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

Programming example

Sobel Edge Detection in UPC

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

Programming example

Distribute data among threads

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

Thread 0
Thread 1
Thread 2
Thread 3

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

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

Programming example

Use upc_forall to do work in parallel

```
upc_forall (i=1; i<N-1; i++; &edge[i][j])
{     for (j=1; j<N-1; j++)
      {
            ...
      }
}
```

Programming example

Sobel Edge Detection– The UPC program

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

Programming example

Notes on the Sobel Example

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

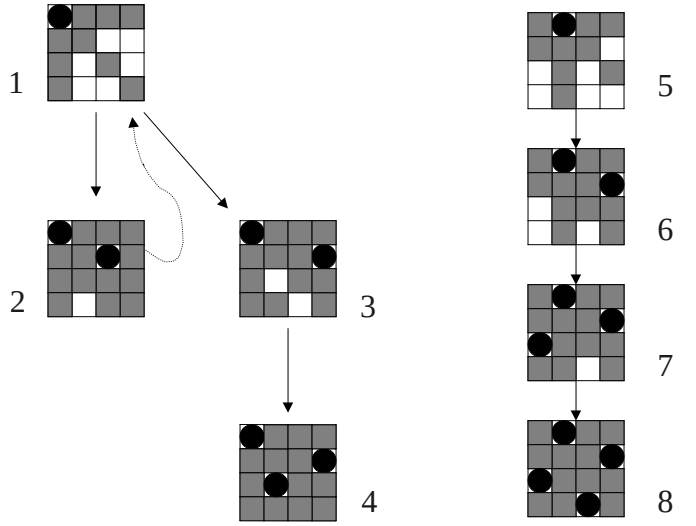
Programming example

Solving the N-Queens Problem in UPC

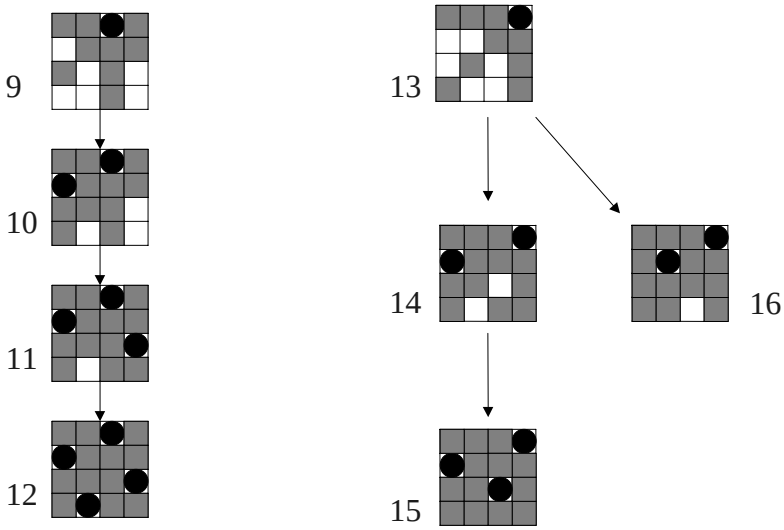
- Place N queens on a NxN chessboard such that no queen is exposed to, and can be killed by, another

Programming example

Example : N=4 (sequential algorithm)



Programming example



Programming example

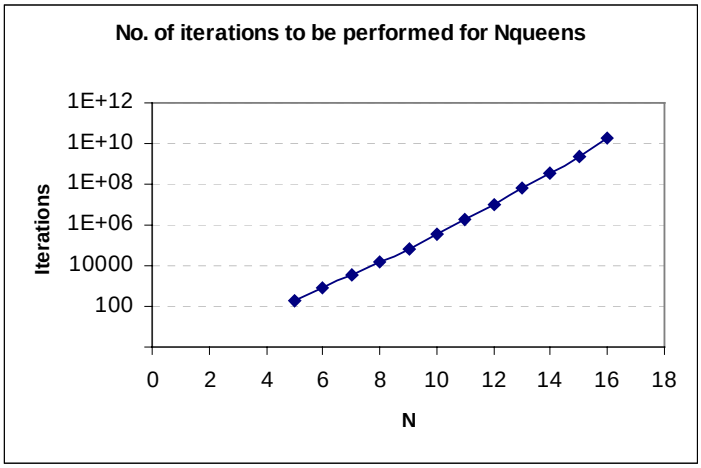


Figure 1. Number of Iterations as a Function the Problem Size (N)

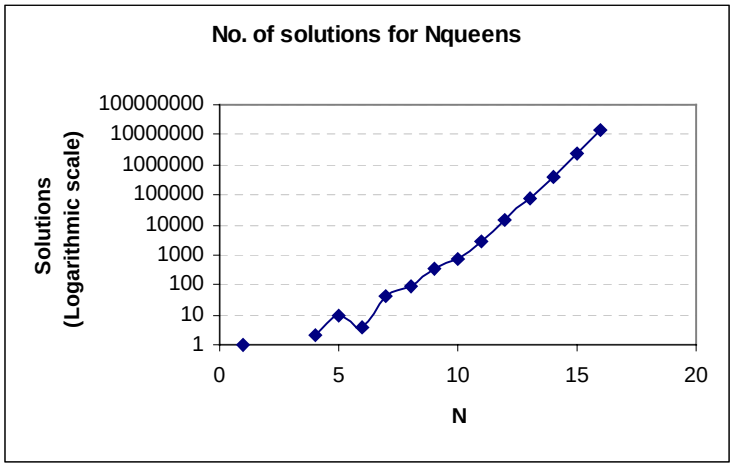
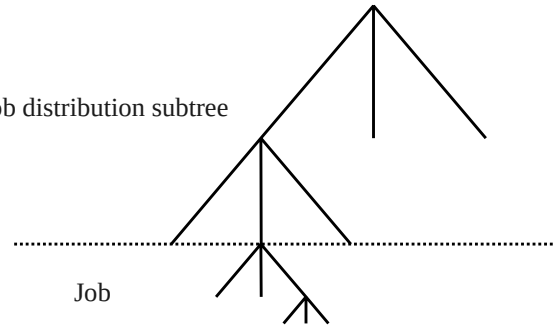


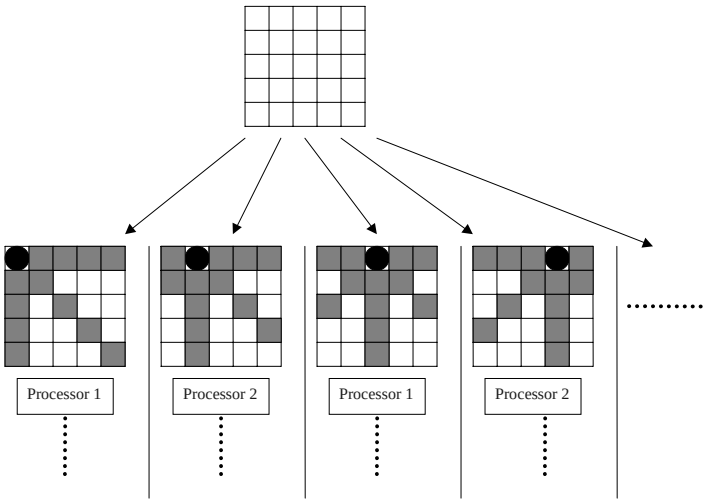
Figure 2. Number of Valid Solutions as a Function of the Problem Size (N)

Parallelizing N-Queens



Programming example

Parallel N-Queens



Programming example

Main program - variables

- shared int number_solns[THREADS];

```
// parameters

shared int n;         // Problem size
shared int l;         // Distribution
shared int method;  // Round-robin / chunking
```

Programming example

Main program - initialization

```
if (MYTHREAD==0) {
    n=atoi(argv[1]);
    if ((n<=0) || (n>16)) {
  fprintf(stderr,"0<n<17\n");
    upc_all_exit () ; // not yet implemented
    }
    l=atoi(argv[2]);
    if ((l<0) || (l>=n)) {
    fprintf(stderr,"0<=l<n\n");
    upc_all_exit () ;
    }
}
```

Programming example

Main program - execution

```
upc_barrier; // make sure thread 0 has set the parameters

number_solns[MYTHREAD] = sched(n,l,method);

upc_barrier; // Complete all solutions before reduction
nsols=0;
if (MYTHREAD==0) {
      for(i=0;i<THREADS;i++)
            nsols+=number_solns[i];
}
```

Programming example

Code for job distribution

```
int sched(int n, int l, int method) {
…
   // Distribution in a round-robin fashion :
   forall(j=0;j<njobs;j++; j) {
         …
         execute sequential algorithm
   }
   // or Distribution in a chunking fashion :
   forall(j=0;j<njobs;j++; (j*THREADS)/njobs ) {
         …
         execute sequential algorithm
   }
}
```

Programming example

Outline

- Design Philosophy
- Background
- Status
- UPC Programming Model
- Introduction to UPC
- Data and pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

Synchronization

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

Synchronization

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 $expr_{opt}$;

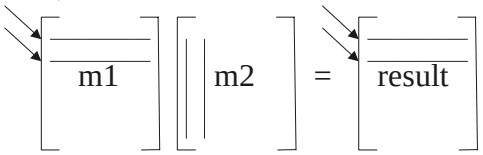
Synchronization

Split-Phase Barrier Example

- Consider computing the following equation:
  $(A + C) * B^2$
  Where A, B, and C are all PxP matrices
- Using the following data structures
  shared[P] int A[P][P]
  shared[P] int C[P][P]
  shared[P] int B[P][P]
  shared[P] int Bsqr[P][P]
  shared[P] int result[P][P]

Synchronization

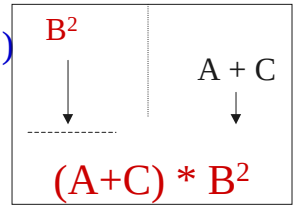
```
matrix_multiplication(shared [P] result [P][P],shared [P] m1 [P][P],
    shared [P] m2 [P][P])
{
upc_forall(i=0;i<P;i++;& m1[i][0])
{
        for (j=0; j<M; j++) {
                int sum=0;
                for(l=0; l< P; l++)
                        sum +=m1[i][l]*m2[l][j];
                result[i][j]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



Synchronization

Computing (A+C) * B^2

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



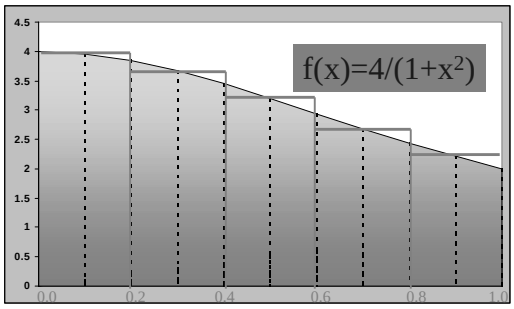
Synchronization

Synchronization - Locks

- In UPC, shared data can be protected against multiple writers :
  - void upc_lock(shared upc_lock_t *l)
  - int upc_lock_attempt(shared upc_lock_t *l) //returns 1 on success and 0 on failure
  - void upc_unlock(shared upc_lock_t *l)

Synchronization

Numerical Integration (computation of π)



$$4\int_0^1 \frac{1}{1+x^2}$$

- Integrate the function f (which equals π)

Synchronization

Example: Using Locks in Numerical Integration

```
// Numerical Integration
// Example – The Famous PI
#include<upc_relaxed.h>
#include<math.h>
#define N 1000000
#define f(x) 1/(1+x*x)

upc_lock_t l;
shared float pi;
void main(void)
{
float local_pi=0.0;
int i;

upc_forall(i=0;i<N;i++; i%THREADS)
      local_pi +=(float) f((.5+i)/(N));
local_pi *= (float) (4.0 / N);
upc_lock(&l);
pi += local_pi;
upc_unlock(&l);
upc_barrier(); // Ensure all is done
if(MYTHREAD==0) printf("PI=%f\n",pi);
}
```

Synchronization

More Locks

```
if (MYTHREAD>THREADS/2)      update_v1();
                    else     update_v2();
```

```
void update_v1()
{   static upc_lock_t l1;

    upc_lock_init(&l1);
    upc_lock(l1);
    v1=expression1(v1);
    upc_unlock(l1);
}
```

```
void update_v2()
{       static upc_lock_t l2;

        upc_lock_init(&l2);
        upc_lock(l2);
        v2=expression2(v2);
        upc_unlock(l2);
}
```

Dynamic lock allocation

- The locks can be managed using the following functions:
- **global lock allocation (à la upc_global_alloc)**
  upc_lock_t * upc_global_lock_alloc(void)
- **collective lock allocation (à la upc_all_alloc)**
  upc_lock_t * upc_all_lock_alloc(void);
- **Lock initialization**
  void upc_lock_init(upc_lock_t *ptr);

Synchronization

Global lock allocation

- **global lock allocation**

  upc_lock_t * upc_global_lock_alloc(void)

- Returns one lock pointer to the calling thread.

Synchronization

Collective lock allocation

- **collective lock allocation**
- upc_lock_t * upc_all_lock_alloc(void);
- Needs to be called by all the threads
- Returns a single lock to all calling threads

Lock initialization

void upc_lock_init(upc_lock_t *ptr);

- Initializes a lock for its usage with upc_lock() or upc_lock_attempt().

Example: Using Locks in Numerical Integration

```
//   Numerical Integration
//   Example – The Famous PI
#include<upc_relaxed.h>
#include<math.h>
#define N 1000000
#define f(x) 1/(1+x*x)

upc_lock_t *l;
shared float pi;
void main(void)
{
float local_pi=0.0;
int i;
l=upc_all_lock_alloc();
```

Allocation

```
upc_forall(i=0;i<N;i++; i%THREADS)
      local_pi +=(float) f((.5+i)/(N));
local_pi *= (float) (4.0 / N);
upc_lock(l);
pi += local_pi;
upc_unlock(l);
upc_barrier(); // Ensure all is done
if(MYTHREAD==0) printf("PI=%f\n",pi);
}
```

Synchronization

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)

Synchronization

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

Synchronization

Memory Consistency Example

```
strict shared int flag_ready = 0;
shared int result0, result1;

switch(MYTHREAD) {
    case 0 : results0 = expression1;
             flag_ready=1; //if not strict, it could be
                     switched with the above statement or
                                executed concurrently
             break;
    case 1 : while(!flag_ready); //Same note
             result1=expression2+results0;
             break;
 }
```

Synchronization

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>

Synchronization

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

Synchronization

Outline

- Design Philosophy
- History
- Background
- Status
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

What are the Opportunities for Performance Enhancements?

- Recognizing local shared accesses (local to thread or to the SMP node) and treating them as private
- Prefetching and aggregating remote accesses
- Overlapping remote accesses with local processing

Performance Tuning

How to Exploit the Opportunities for Performance Enhancement?

- Compiler optimizations
- Run-time system
- Hand tuning

Performance Tuning

How can you Optimize the Performance of your Code?

- Become familiar with the UPC specific compiler optimization options
- Be aware of whether the vendor has a run time system that can help optimizing and how to set respective environment variables
- When all does not get you what you want, try the following hand tuning techniques

Performance Tuning

List of Hand Tunings for UPC Code Optimization

- Use local pointers instead of shared pointers when dealing with local shared data, through casting and assignments
- Use block copy instead of copying elements one by one with a loop, through string operations or structures
- Overlap remote accesses with local processing using split-phase barriers

Performance Tuning

Typical Performance of Shared vs. Private Accesses

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

Performance Tuning

Using Local Pointers Instead of Shared Pointers

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

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

Performance Tuning

Aggregating Remote Accesses using Block Copy

```
// Copying the array element by element
shared [] int a[1000], b[1000];
for (j=1; j<1000; j++)
    b[j]=a[j];


// Copying the whole array at once using structures
typedef struct { int array[1000] } st_copy;
shared [] int a[1000], b[1000];
st_copy *p_a, *p_b;

p_a =(st_copy*) a; p_b = (st_copy*)b;
*p_a = *p_b;
```

Performance Tuning

Aggregating Remote Accesses with String Functions

This is not yet available on any UPC compilers

```
shared [] int a[1000], b[1000];

// Copy element by element
for (j=1; j<1000; j++)
    b[j]=a[j];

// Copy the whole array at once using string functions
upc_memcpy(b, a , sizeof(*a)*1000);
```

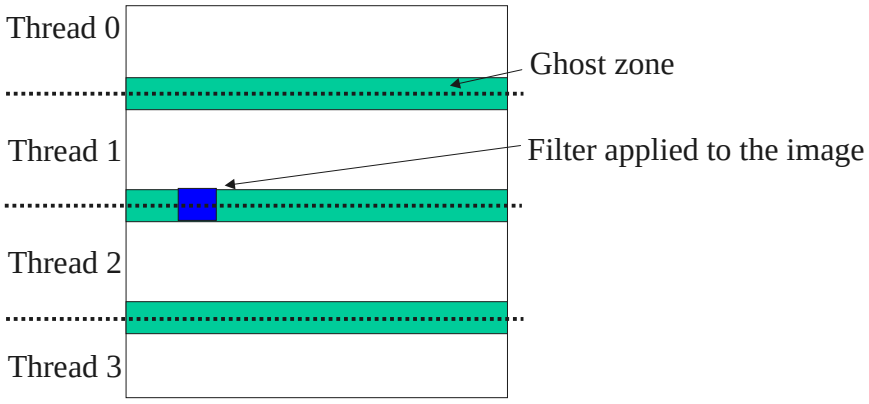
Performance Tuning

Matrix Multiplication with Block Copy

```
// mat_mult_3.c

#include <upc_relaxed.h>

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

int b_local[P][M];

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

        upc_memget(b_local, b, P*M*sizeof(int));

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

Performance Tuning

Overlapping Remote Accesses with Local Processing



Overlapping Remote Accesses with Local Processing

```
upc_memcpy(ghost_copy, ghost_zone, size);
upc_notify;
// work on everything but the ghost_zones
upc_wait;
// work with the ghost_zones
```

Runtime optimizations on Compaq (1)

- Caching: The runtime system can cache the shared data to eliminate some remote fetches.
- List of the related environment variables and (their default values):
  - UPCRTS_USE_CACHE (false)
  - UPCRTS_CACHE_SETS (128)
  - UPCRTS_CACHE_BLOCK_SIZE (64)
  - UPCRTS_CACHE_ASSOCIATIVITY (4)
  - UPCRTS_DISP_CACHE_STATISTICS (False)

Runtime optimizations on Compaq (2)

- Prefetching: The runtime system can prefetch the shared data to eliminate some remote fetches.
- List of the related environment variables and (their default values):
  - UPCRTS_USE_PREFETCH (False)
  - UPCRTS_PREFETCH_DISTANCE (3)
  - UPCRTS_PREFETCH_TRAINING (20)
  - UPCRTS_DISP_PREFETCH_STATISTICS (False)

How to set an environment variable

- With sh-like shell:
  - export UPCRTS_USE_CACHE=*value*
- With csh shell:
  - setenv UPCRTS_USE_CACHE *value*

SMP local optimization

- Allows threads to access the shared memory of any other thread in the same node as private
- Compilation option (-smp_local)
- Pros:
  - Eliminate the communication when accessing shared memory of other threads in the same SMP node
- Cons:
  - Dynamically allocate shared static objects

Outline

- Design Philosophy
- Background
- Status
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

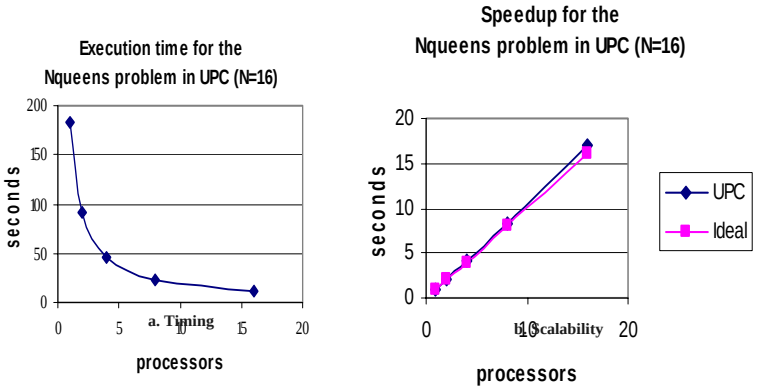
Outline

Performance of UPC

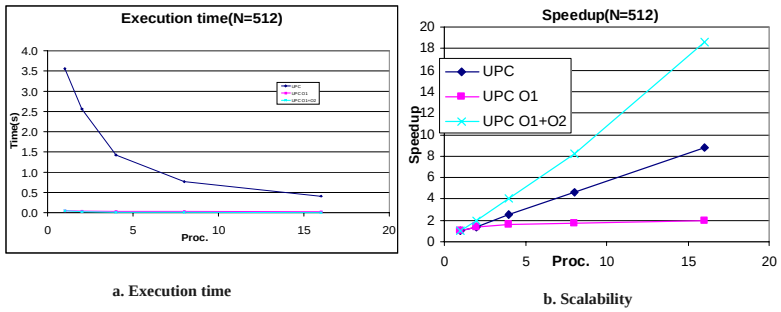
- Current benchmarking results on Compaq for:
  - Nqueens Problem
  - Matrix Multiplications
  - Sobel Edge detection
  - Synthetic Benchmarks
- Check the web site for a report with extensive measurements on Compaq and T3E

Performance

Performance of Nqueens on the Compaq AlphaServer

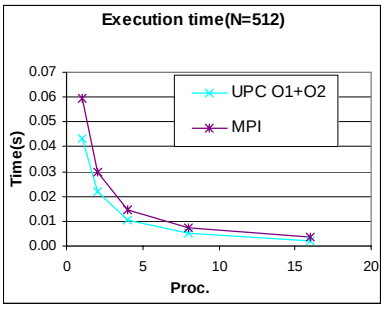


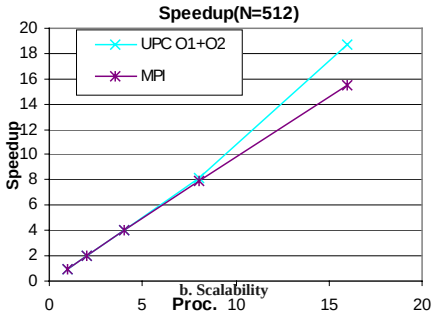
Performance of Edge detection on the Compaq AlphaServer SC



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

Performance of Optimized UPC versus MPI for Edge detection

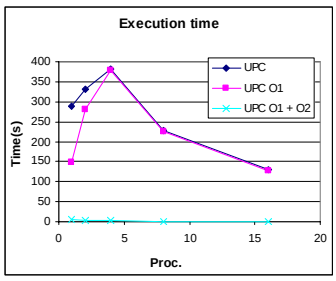




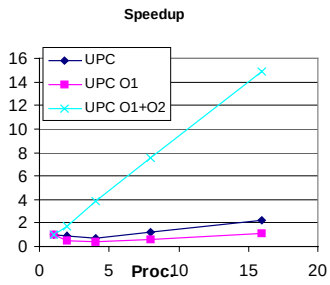
b. Scalability

Performance

Effect of Optimizations on Matrix Multiplication on the AlphaServer SC



a. Execution time

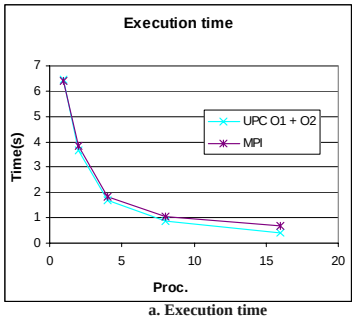


b. Scalability

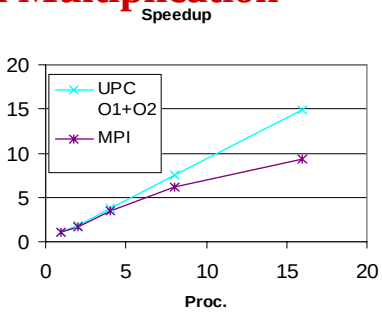
O1: using private pointer instead of shared pointer
O2: using structure copy instead of element by element

Performance

Performance of Optimized UPC versus C + MPI for Matrix Multiplication



a. Execution time



b. Scalability

Performance

Outline

- Design Philosophy
- Background
- Status
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

Conceptual Comparison of UPC, OpenMP, MPI, and HPF:

Comparison

Features Matrix

	MPI	HPF	OpenMP	UPC	Remarks
Programming model	Message Passing	Data Parallel	Shared Memory	Distributed Shared Memory	--
Expressing Parallelism	Library	F90 Extension + Directives	Library Directives	C extension	--
Language Specificity	C and Fortran bindings	Fortran	C and Fortran bindings	C	--
Architectures Supported	MIMD	MIMD and SIMD	SMP	MIMD	--
Degree of Parallelism	Fixed in execution command (MPI-1)	Virtual Processes fixed in the code and # of physical processors in the execution command or through environment setting	Set by default, environment, or from inside the code (inversely ordered according to precedence). Can change during execution.	Fixed at compile time. New implementations are free to specify it at run time.	

Features Matrix

	MPI	HPF	OpenMP	UPC	Remarks
Nested Parallelism	Not supported	Not Supported	Supported (Fork/Join)	Not supported	
Incremental Parallelisation of code	Indirectly supported	No	Yes	Indirectly supported	
Locality Exploitation	Possible but hard (data distribution using explicit communication)	Yes through alignment & distribution	No	Yes (Blocking and affinity)	

Features Matrix

	MPI	HPF	OpenMP	UPC	Remarks
Effect of Parallelism on variable attributes	N/A	N/A	Global variables are shared by default Children threads share parent variables	N/A	Better protection of global data in UPC
Collective Operations	Broadcast Scatter Gather Reductions	Broadcast, Scatter and Gather implied in language Reduction implemented	Same as HPF	Same as HPF, but with no reduction in current specs	Reduction needed in UPC. No convenient way of implementing the tree-like O(log n) reduction by the average programmer.

Features Matrix

	MPI	HPF	OpenMP	UPC	Remarks
Work Sharing	Laborious	N/A (inherited from data parallel)	Worksharing with several scheduling options	Different affinity values in upc_forall	Specify more UPC options that can Achieve static load balancing Dynamic load balancing
Data Distribution and allocation Declarations	NO	Various directives for blocked distribution of data	N/A	Blocked array declarations	.
Heterogeneity support	Strong MPI data typing	N/A	N/A	N/A	

Features Matrix

	MPI	HPF	OpenMP	UPC	Actions /Remarks
Memory Consistency Model Control	N/A	N/A (inherited from data parallel)	Strict only	Strict and relaxed allowed on global or next statement level.	
Virtual Topologies	Supported	Limited Support	N/A	N/A	.
Dynamic Memory Allocation	Private only	Private	Variable as per fork/join in OpenMP	Private or shared with or without blocking	

Features Matrix

	MPI	HPF	OpenMP	UPC	Remarks
Synchronization	Barriers	N/A (synchronous data parallel)	Via critical sections, atomic operations, barrier, master section	Barriers, split phase barrier, locks, and memory consistency control	
Type Conversion	Depends on language used	Automatic between data types	Language based casting-like clauses	C rules Casting of shared pointers to private pointers	.
Pointers To Shared Space	N/A	N/A	Same as variables	Yes	

Outline

- Design Philosophy
- Background
- Status
- UPC Highlights and Programming Model
- Introduction to UPC
- Data and Pointers
- Dynamic Memory Allocation
- Programming Examples

9. Synchronization
10. Performance Tuning
11. Performance Results
12. Conceptual Comparison of UPC, MPI, OpenMP and HPF
13. Concluding Remarks

Outline

Conclusions

- UPC is easy to program in for C writers, significantly easier than alternative paradigms at times
- UPC exhibits very little overhead when compared to 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 compared favorably with MPI on the Compaq AlphaServer

- Hand tuning may sometimes make the code a little bit like MPI
- As compiler optimizations effectively exploit the opportunities for improvements, as discussed, there will be no need for hand tuning. There is some evidence with Compaq that we could get there

- UPC and the distributed shared memory programming model combines the ease of shared memory programming and the power of message passing and is likely to be the way of the future, should compilers effectively exploit optimization opportunities.
- Even if this does not happen, although it should happen, UPC will provide an alternative to MPI which for many problems will be easier to use and better in performance with little tuning.

Conclusions