PACT 08

Productive Parallel Programming in PGAS

Calin Cascaval - IBM TJ Watson Research Center
Gheorghe Almasi - IBM TJ Watson Research Center
Ettore Tiotto - IBM Toronto Laboratory
Kit Barton - IBM Toronto Laboratory

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Outline

1. Overview of the PGAS programming model
2. Scalability and performance considerations
3. Compiler optimizations
4. Examples of performance tuning
5. Conclusions

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

1. Overview of the PGAS programming model

Some slides adapted with permission from Kathy Yelick

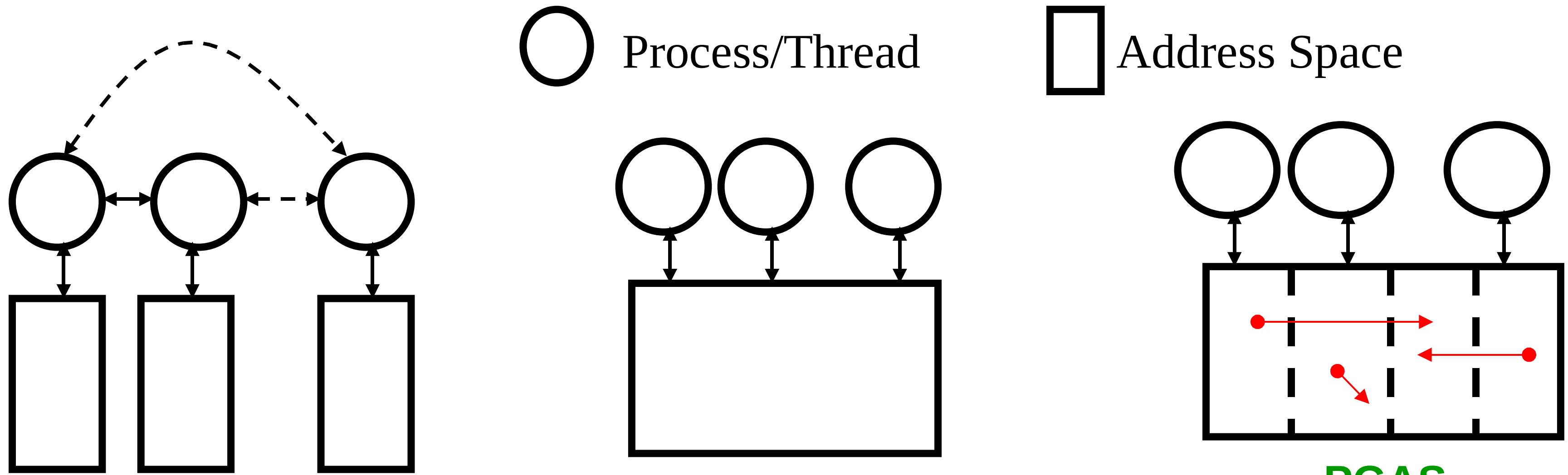
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Partitioned Global Address Space

- **Explicitly parallel, shared-memory like programming model**
- **Global addressable space**
  - Allows programmers to declare and “directly” access data distributed across the machine
- **Partitioned address space**
  - Memory is logically partitioned between *local* and *remote* (a two-level hierarchy)
  - Forces the programmer to pay attention to data locality, by exposing the inherent NUMA-ness of current architectures
- **Single Processor Multiple Data (SPMD) execution model**
  - All threads of control execute the **same** program
  - Number of threads fixed at startup
  - Newer languages such as X10 escape this model, allowing fine-grain threading
- **Different language implementations:**
  - **UPC** (C-based), CoArray Fortran (Fortran-based), Titanium and X10 (Java-based)

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Partitioned Global Address Space



Message passing

MPI

Shared Memory

OpenMP

PGAS

UPC, CAF, X10

- **Computation is performed in multiple places.**
- **A place contains data that can be operated on remotely.**
- **Data lives in the place it was created, for its lifetime.**
- **A datum in one place may point to a datum in another place.**
- **Data-structures (e.g. arrays) may be distributed across many places.**
- **Places may have different computational properties (mapping to a hierarchy of compute engines)**

A place expresses locality.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Overview and Design Philosophy

- **Unified Parallel C (UPC) is:**
  - An explicit parallel extension of ANSI C
  - A partitioned global address space 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**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Execution Model

- **A number of threads working independently in a SPMD fashion**
  - Number of threads 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 (compiler option)
    - The program may use THREADS as a compile-time constant
    - The compiler generates more efficient code
  - **Dynamic threads mode:**
    - Compiled code may be run with varying numbers of threads
    - THREADS is specified at runtime time by the user (via env. variable)

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Hello World in UPC

- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with N threads, it will run N copies of the program (Single Program executed by all threads).

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

int main() {
  printf("Thread %d of %d: Hello UPC world\n", MYTHREAD,
THREADS);
  return 0;
}
```

```
hello > xlupc helloWorld.upc
hello > env UPC_NTHREADS=4 ./a.out
Thread 1 of 4: Hello UPC world
Thread 0 of 4: Hello UPC world
Thread 3 of 4: Hello UPC world
Thread 2 of 4: Hello UPC world
```

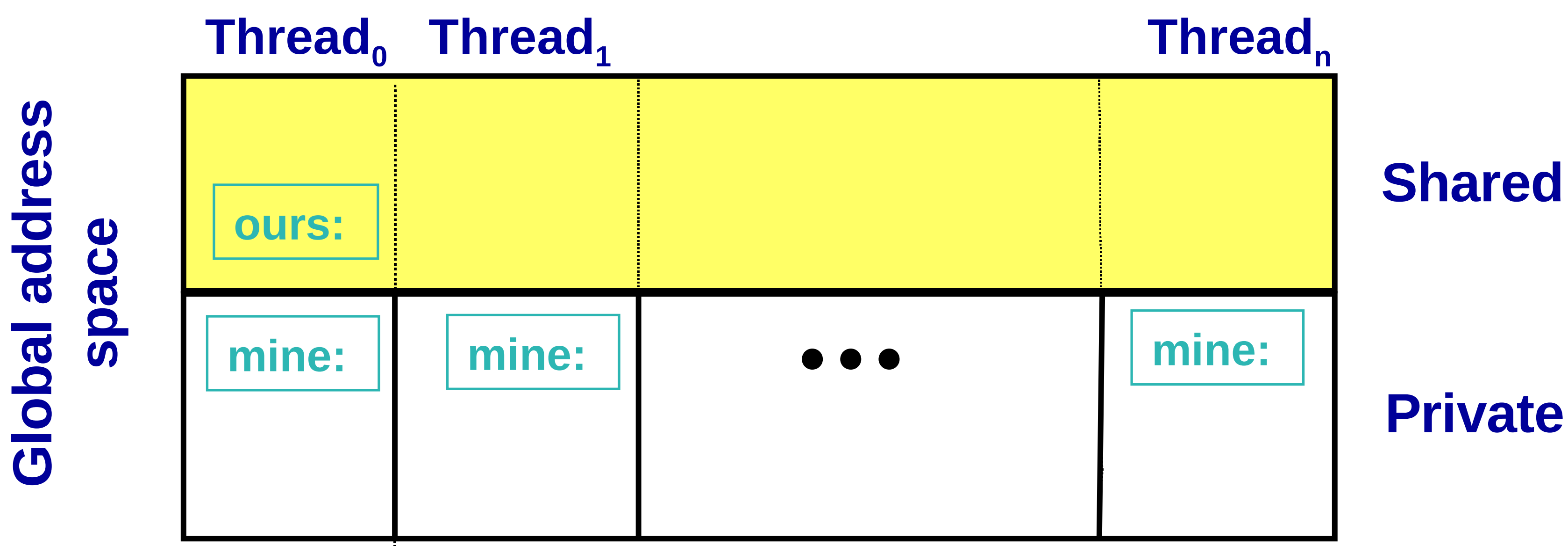
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

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;
```

- **Shared variables may not be declared automatic, i.e., may not occur in a in a function definition, except as static. Why?**



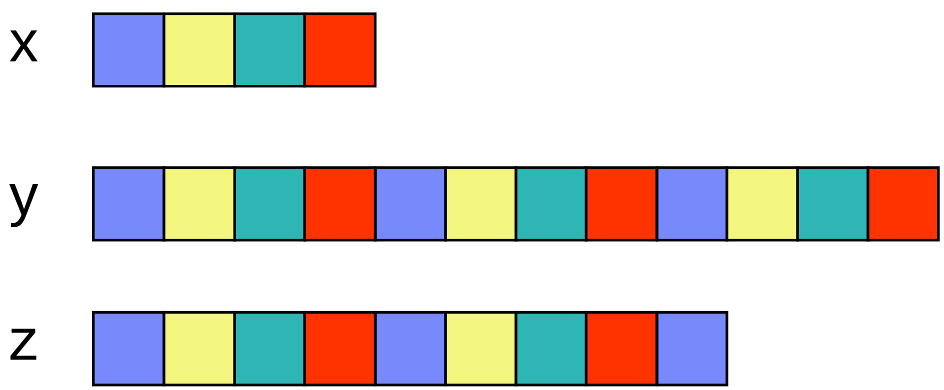
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Shared Arrays

- **Shared arrays are spread over the threads, distributed in a cyclic fashion**

```
shared int x[THREADS];          /* 1 element per thread */
shared int y[3][THREADS];       /* 3 elements per thread */
shared int z[3][3];             /* 2 or 3 elements per thread */
```

- **Assuming THREADS = 4:**



Think of a linearized C array, then map round-robin on THREADS

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Work Sharing with upc_forall()

- **The owner computes rule is very common in parallel programming**
  - Loop over all; work on those owned by this proc
- **UPC adds a special type of loop**

```
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 on each thread. It may have one of two types:**
  - **Integer:** `affinity%THREADS` is `MYTHREAD`

    ```
    upc_forall(i=0; i<N; ++i; i)
    ```

  - **Pointer:** `upc_threadof(affinity)` is `MYTHREAD`

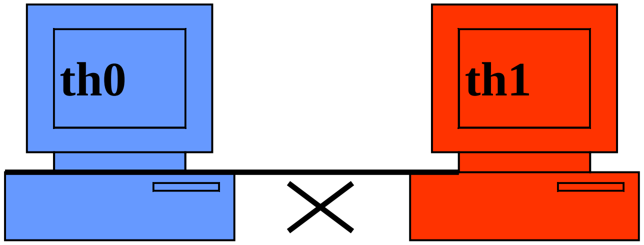
    ```
    upc_forall(i=0; i<N; ++i; &A[i])
    ```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Work Sharing with upc_forall()

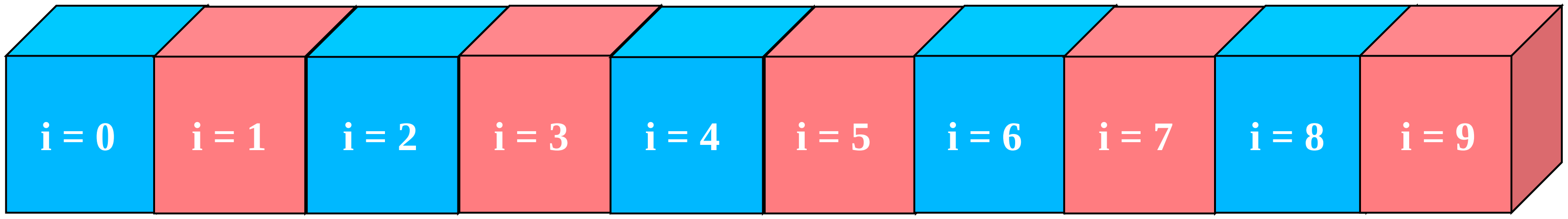
- Similar to C for loop, 4th field indicates the affinity
- Thread that "owns" elem. A[i] executes iteration

```
shared int A[10],B[10],C[10];
upc_forall(i=0; i < 10; i++; &A[i]) {
    A[i] = B[i] + C[i];
}
```



No communication

2 threads



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Vector Addition with upc_forall

```
#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];
}
```

- 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.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Global Synchronization

- **Controls relative execution of threads**
- **UPC has two basic forms of barriers:**
  - Barrier: block until all other threads arrive
    `upc_barrier`
  - Split-phase barriers
    `upc_notify;` this thread is ready for barrier
    do computation unrelated to barrier
    `upc_wait;` wait for others to be ready
- **Optional labels allow for debugging**

```
#define MERGE_BARRIER 12
if (MYTHREAD%2 == 0) {
   ...
   upc_barrier MERGE_BARRIER;
} else {
   ...
   upc_barrier MERGE_BARRIER;
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Recap: Shared Variables, Work sharing and Synchronization

- **With what you've seen until now, you can write a bare-bones data-parallel program**
- **Shared variables are distributed and visible to all threads**
  - Shared scalars have affinity to thread 0
  - Shared arrays are distributed (cyclically by default) on all threads
  - We shall look next at how to control memory layouts, shared pointers, etc.
- **Execution model is SPMD with the `upc_forall` provided to share work**
- **Barriers and split barriers provide global synchronization**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Blocked Layouts in UPC

- **The cyclic layout is typically stored in one of two ways**
  - Distributed memory: each processor has a chunk of memory
    - Thread 0 would have: 0,THREADS, THREADS*2,… in a chunk
  - Shared memory machine: all data may be on one chunk
    - Shared memory would have: 0,1,2,…THREADS,THREADS+1,…
- **Vector addition example can be rewritten as follows**

```
#define N 100*THREADS
shared int [*] v1[N], v2[N], sum[N];
void main() {
   int i;
   upc_forall(i=0; i<N; i++; &a[i])
        sum[i]=v1[i]+v2[i];
}
```

blocked layout

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Layouts in General

- **All non-array objects have affinity with thread zero.**
- **Array layouts are controlled by layout specifiers:**
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [0] or [] (indefinitely layout, all on 1 thread)
  - [b] or [b1][b2]…[bn] = [b1*b2*…bn] (fixed block size)
- **The affinity of an array element is defined in terms of:**
  - block size, a compile-time constant
  - and THREADS.
- **Element i has affinity with thread**

  (i / block_size) % THREADS

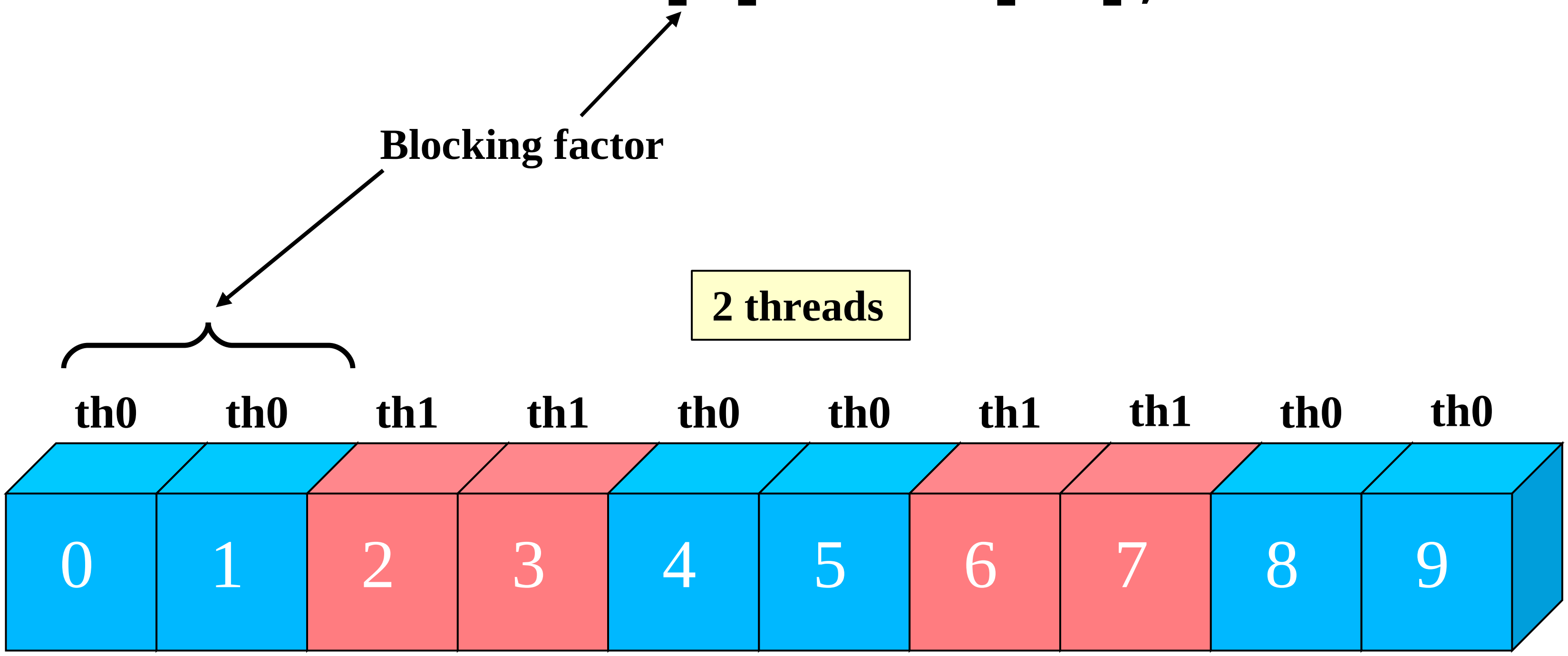
- **In 2D and higher, linearize the elements as in a C representation, and then use above mapping**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Distribution of a shared array in UPC

- Elements are distributed in block-cyclic fashion
- Each thread “owns” blocks of adjacent elements

```
shared [2] int X[10];
```



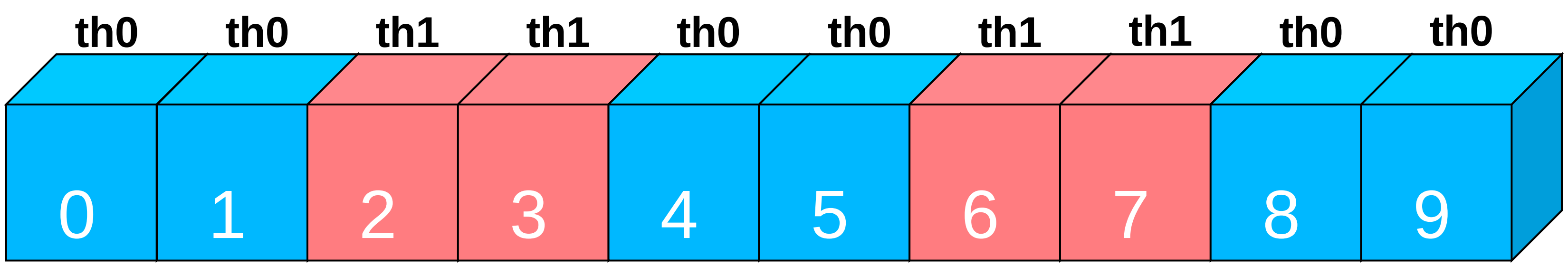
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Physical layout of shared arrays

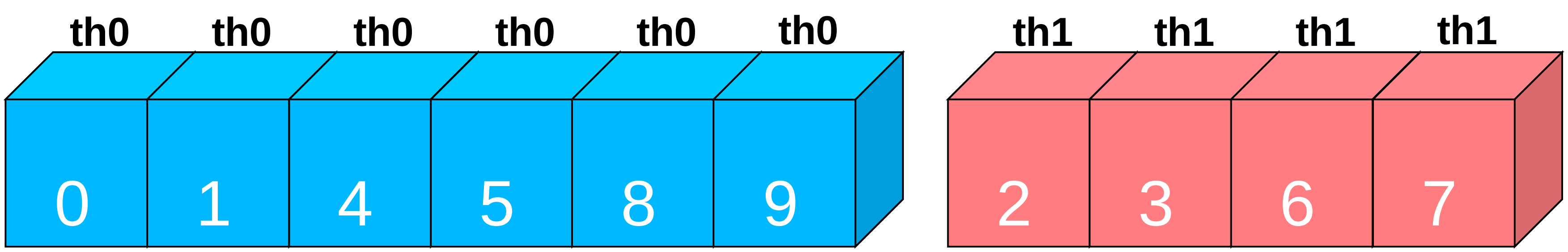
```
shared [2] int X[10];
```

2 threads

Logical Distribution



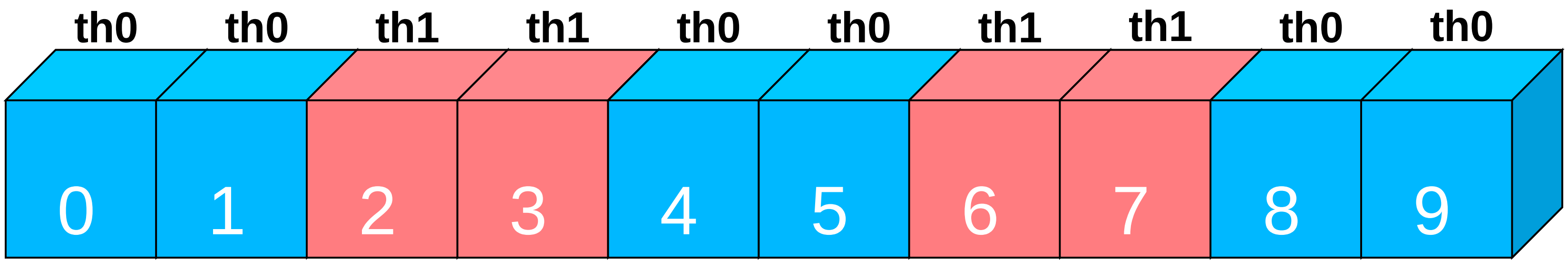
Physical Distribution



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Terminology

```
shared [2] int X[10];
```



- **upc_threadof(&a[i])**
  - Thread that owns a[i]
- **upc_phaseof(&a[i])**
  - The position of a[i] within its block
- **course(&a[i])**
  - The block index of a[i]

Examples

upc_threadof(&a[2]) = 1

upc_threadof(&a[5]) = 0

upc_phaseof(&a[2]) = 0

upc_phaseof(&a[5]) = 1

course(&a[2]) = 0

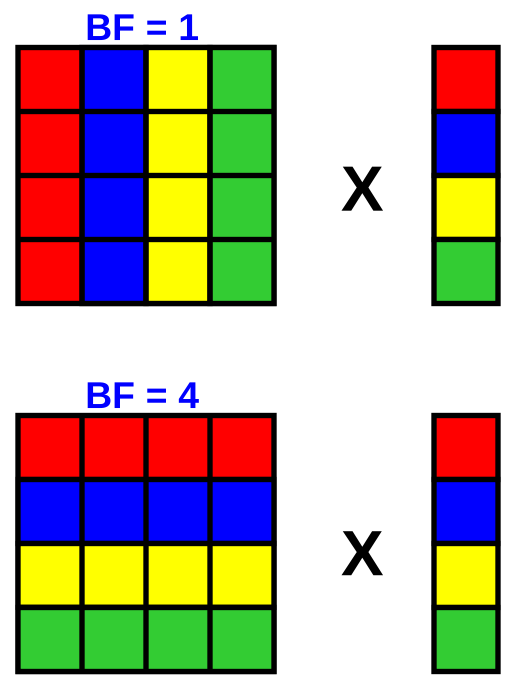
course(&a[5]) = 1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Matrix Vector Multiplication Code

- **Matrix-vector multiplication with matrix stored by rows**

```
#define N   THREADS
shared [BF] int a[N][N];
shared int b[N], c[N];


int main (void) {
  int i, j;
  upc_forall( i = 0; i < N; i++; i) {
    c[i] = 0;
    for (j = 0; j < N; j++)
      c[i] += a[i][j]*b[j];
  }
  return 0;
}
```

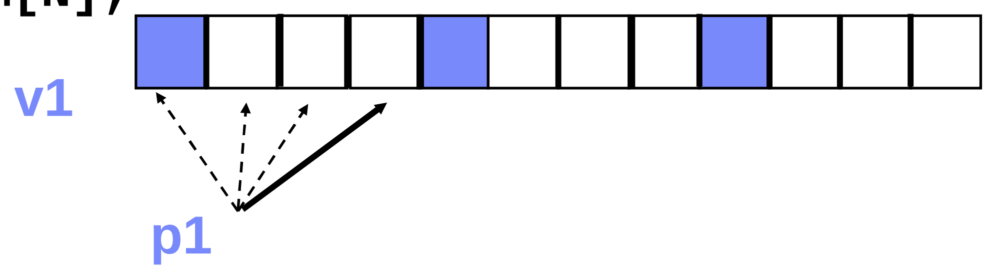


This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Pointers to Shared vs. Arrays

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

```
#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;

   }
}
```



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Pointers

Where does the pointer point to?

Where does the pointer reside?

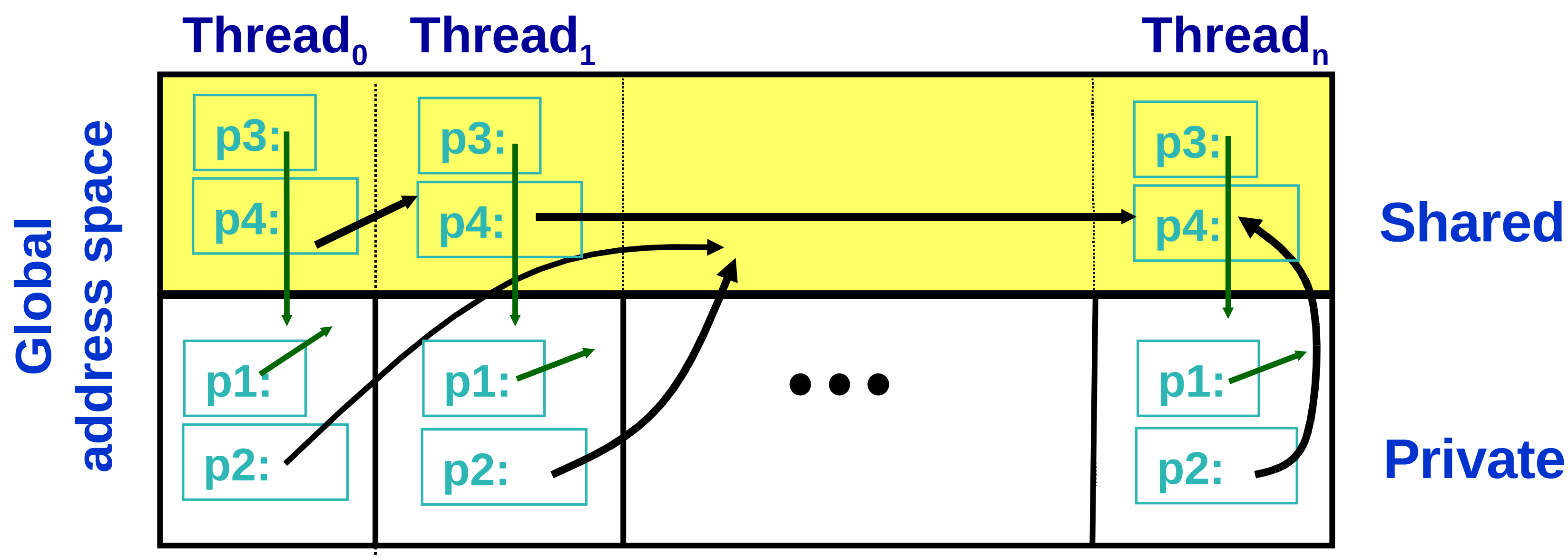
	Local	Shared
Private	PP (p1)	PS (p3)
Shared	SP (p2)	SS (p4)

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

Shared to private is not recommended. Why?

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

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.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Common Uses for UPC Pointer Types

```
int *p1;
```

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

```
shared int *p2;
```

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

```
int * shared p3;
```

- Not recommended

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

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

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

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**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Pointer Query 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

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Synchronization primitives

- We have seen upc_barrier, upc_notify and upc_wait
- UPC supports locks:
  - Represented by an opaque type: `upc_lock_t`
  - Must be allocated before use:

```
upc_lock_t *upc_all_lock_alloc(void);
```

allocates 1 lock, pointer to all threads

```
upc_lock_t *upc_global_lock_alloc(void);
```

allocates 1 lock, pointer to all threads

  - To use a lock:

```
void upc_lock(upc_lock_t *l)
void upc_unlock(upc_lock_t *l)
```

use at start and end of critical region

  - Locks can be freed when not in use

```
void upc_lock_free(upc_lock_t *ptr);
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Dynamic memory allocation

- **As in C, memory can be dynamically allocated**
- **UPC provides several memory allocation routines to obtain space in the shared heap**
  - shared void* upc_all_alloc(size_t nblocks, size_t nbytes)
    - a collective operation that allocates memory on all threads
    - layout equivalent to: shared [nbytes] char[nblocks * nbytes]
  - shared void* upc_global_alloc(size_t nblocks, size_t nbytes)
    - A non-collective operation, invoked by one thread to allocate memory on all threads
    - layout equivalent to: shared [nbytes] char[nblocks * nbytes]
  - shared void* upc_alloc(size_t nbytes)
    - A non-collective operation to obtain memory in the thread's shared section of memory
  - void upc_free(size_t nbytes)
    - A non-collective operation to free data allocated in shared memory
    - Why do we need just one version?

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Distributed arrays allocated dynamically

```
typedef shared [] int *sdblptr;
shared sdblptr directory[THREADS];

int main() {
  …
  directory[MYTHREAD] = upc_alloc(local_size*sizeof(int));
  upc_barrier;
  …
  /* use the array */
  upc_barrier;
  upc_free(directory[MYTHREAD]);
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Data movement

- Fine grain (array element, by array element access) are easy to program in an imperative way. However, especially on distributed memory machines, block transfers are more efficient
- UPC provides library functions for data movement and collective operations:
  - upc_memset
    - Set a block of values in shared memory
  - upc_memget, upc_memput
    - Transfer blocks of data from shared memory to/from private memory
  - upc_memcpy
    - Transfer blocks of data from shared memory to shared memory
- Collective operations (broadcast, reduce, etc.)
    - A set of function calls is specified in the standard, but it’s being reworked. More about this a bit later

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Memory Consistency

- The consistency model defines the order in which one thread may see another threads accesses to memory
  - If you write a program with unsychronized accesses, what happens?
  - Does this work?

```
data = …                    while (!flag) { };
flag = 1;                   … = data;   // use the data
```

- UPC has two types of accesses:
  - Strict: will always appear in order
  - Relaxed: may appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:

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

  - Which makes all accesses in the file relaxed by default
  - Use strict on variables that are used as synchronization (**flag**)

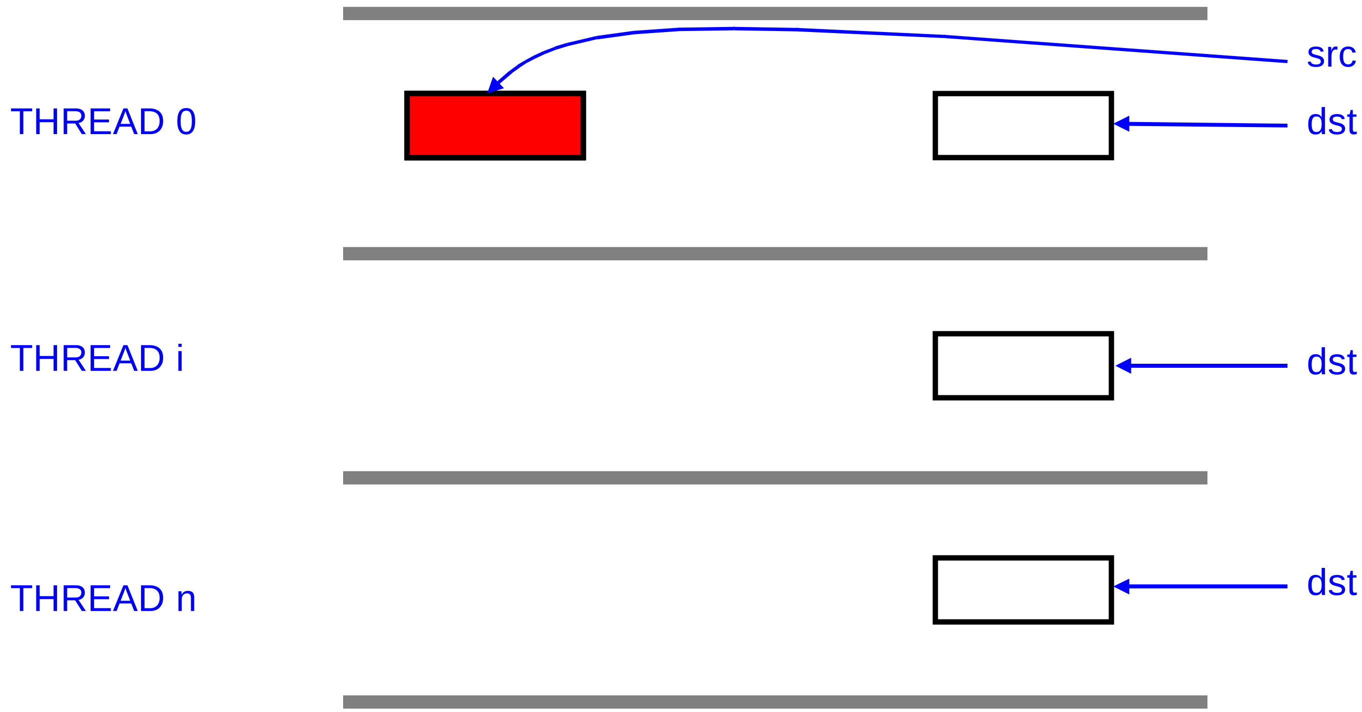
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Data movement Collectives

- Used to move shared data across threads:
  - upc_all_broadcast(shared void* dst, shared void* src, size_t nbytes, …)
    - A thread copies a block of memory it "owns" and sends it to all threads
  - upc_all_scatter(shared void* dst, shared void *src, size_t nbytes, …)
    - A single thread splits memory in blocks and sends each block to a different thread
  - upc_all_gather(shared void* dst, shared void *src, size_t nbytes, …)
    - Each thread copies a block of memory it "owns" and sends it to a single thread
  - upc_all_gather_all(shared void* dst, shared void *src, size_t nbytes, …)
    - Each threads copies a block of memory it "owns" and sends it to all threads
  - upc_all_exchange(shared void* dst, shared void *src, size_t nbytes, …)
    - Each threads splits memory in blocks and sends each block to all thread
  - upc_all_permute(shared void* dst, shared void *src, shared int* perm, size_t nbytes, …)
    - Each threads copies a block of memory and sends it to thread in perm[i]

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

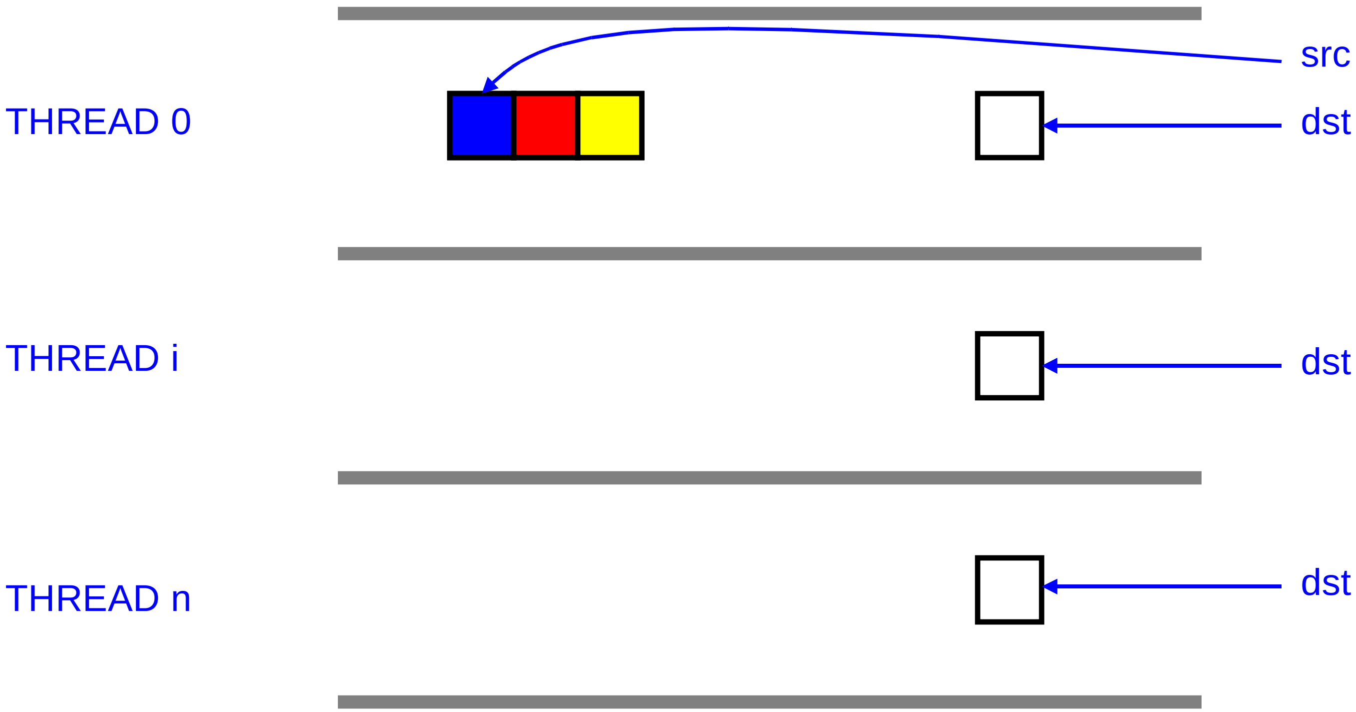
upc_all_broadcast



Thread 0 copies a block of memory and sends it to all threads

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

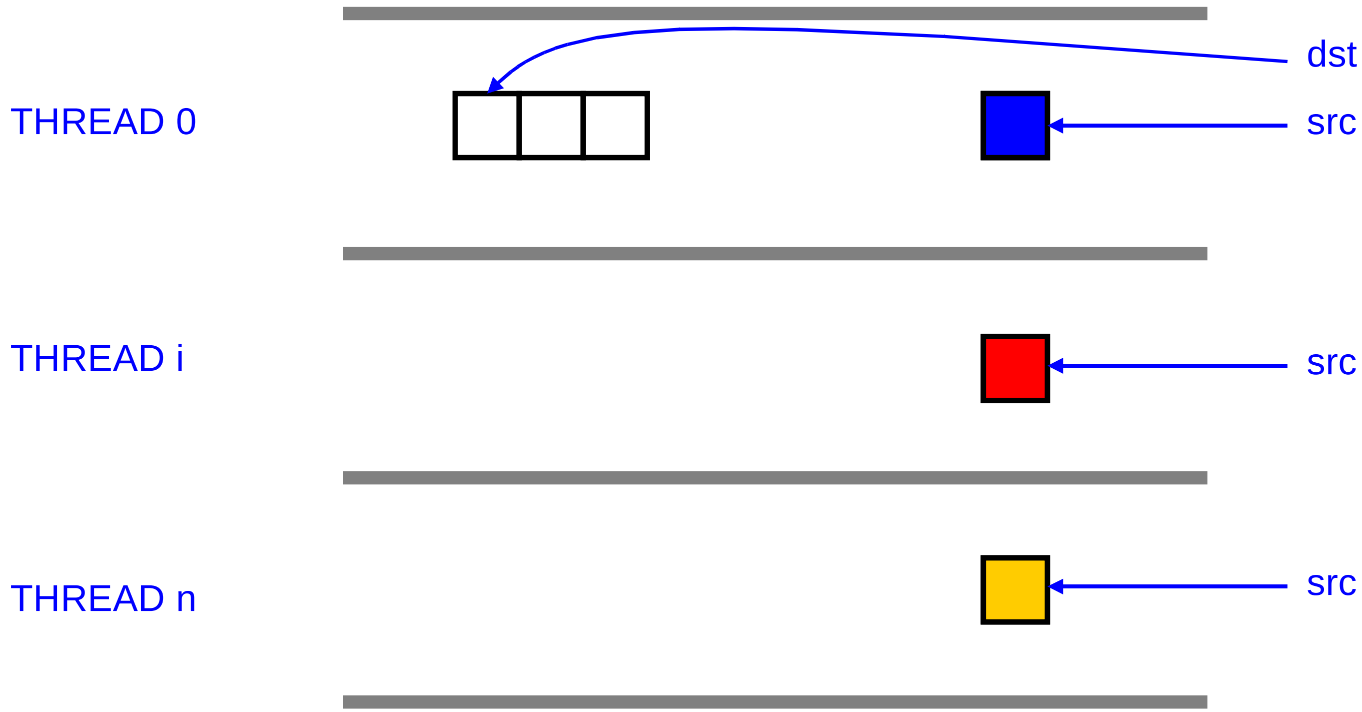
upc_all_scatter



Thread 0 sends a unique block to all threads

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

upc_all_gather



Each thread sends a block to thread 0

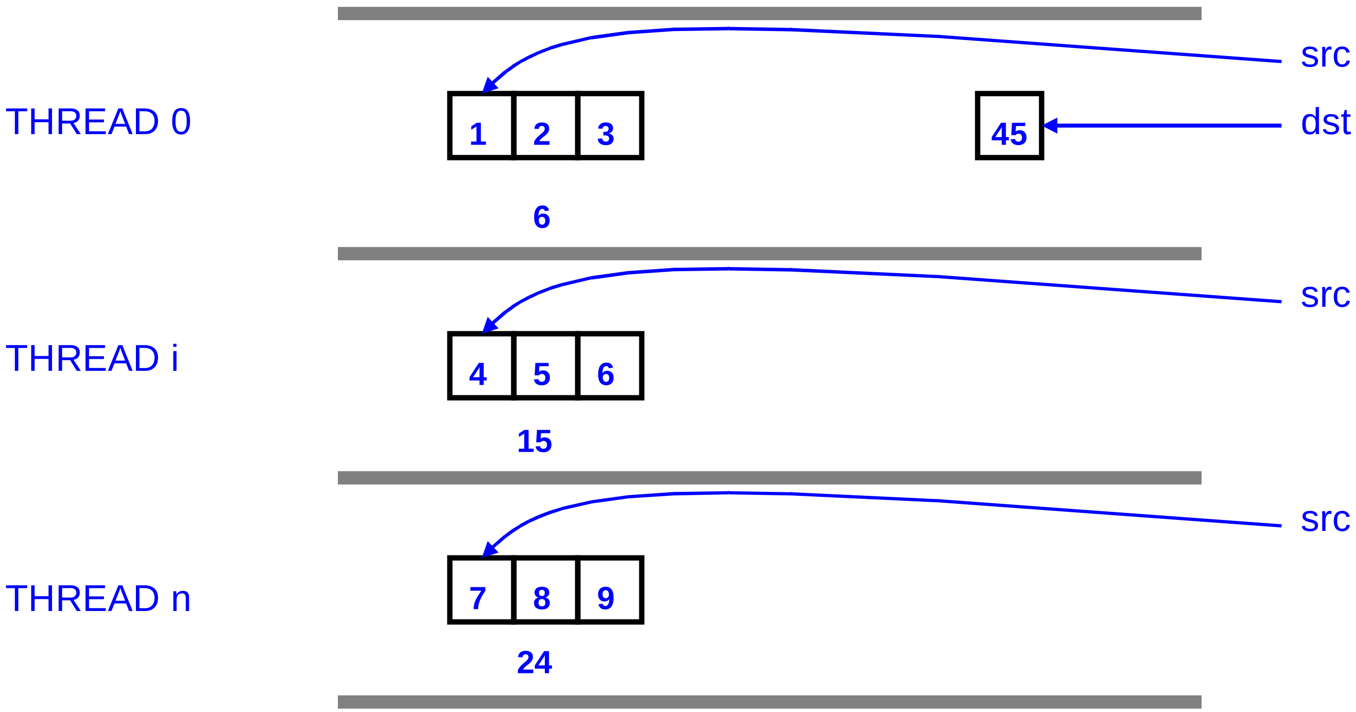
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Computational Collectives

- Used to perform data reductions
  - upc_all_reduceT(shared void* dst, shared void* src, upc_op_t op, …)
  - upc_all_prefix_reduceT(shared void* dst, shared void *src, upc_op_t op, …)
- One version for each type T (22 versions in total)
- Many operations are supported:
  - OP can be: +, *, &, |, xor, &&, ||, min, max
  - perform OP on all elements of src array and place result in dst array

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

upc_all_reduce



Threads perform partial sums, each partial sum added and result stored on thread 0

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

PACT 08

Scalability and performance considerations

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Scalability: Rules of thumb

- **Things to avoid:**
  - UPC tempts user into fine-grain communication
  - UPC tempts user into bad data layouts
  - The siren song of UPC locks
- **Things to take advantage of:**
  - Global view makes reasoning about program easier
    - The “G” ins PGAS
  - Collective communication

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Simple sum: Introduction

- Simple forall loop to add values into a sum

```
shared int values[N], sum;

sum = 0;
upc_forall (int i=0; i<N; i++; &values[i])
   sum += values[i];
```

- Is there a problem with this code ?
- Implementation is broken: “sum” is not guarded by a lock
- Write-after-write hazard; will get different answers every time

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Simple sum: using locks

- Easy you say … let’s use a lock !

```
shared int values[N], sum;
upc_lock_t mylock;

upc_forall (int i=0; i<N; i++; &values[i]) {
        upc_lock (&mylock);
        sum += values[i];
        upc_unlock (&mylock);
}
```

- Correct implementation ☺☺☺
- But horrible performance ☹☹☹
- Lock is used for every array value !!!

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Simple sum: minimizing lock use

- Better performance if N >> THREADS
- Still O(N) communication!

```
shared int values[N], sum;
shared int partialsums[THREADS];

partialsum[MYTHREAD]=0;
upc_forall (int i=0; i<N; i++; &values[i]) {
  partialsum[MYTHREAD] += values[i];
}

upc_forall (int i=0; i<THREADS; i++; partialsums[i]) {
  upc_lock (&mylock);
  sum += partialsums[i];
  upc_unlock (&mylock);
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Simple sum: avoiding locks

- **Assuming N = k * THREADS (or array padded with zeroes)**

```
shared int values[N], sum;

upc_all_reduceI (&sum,
                 values,
                 UPC_ADD,
                 THREADS,
                 N/THREADS,
                 NULL,
                 UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

- **Typical O(log(n)) scalability (like MPI reductions)**
- **Your lesson: avoid locks! There is almost always a better solution**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

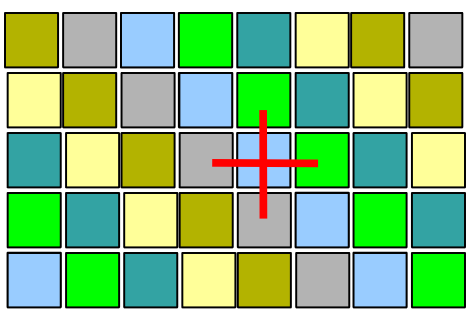
Access Granularity: Stencil

Communication traffic:
4 * N * N * THREADS elements
4 * N * N * THREADS accesses

Naive solution:

```
shared double A[N][N];

upc_forall (i=1; i<N-1; i++; continue)
  upc_forall (j=1; j<N-1; j++; &A[i][j])
    A[i][j] = 0.25 *(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1];
```

This is bad because all accesses right of 0.25 are likely non-local.



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

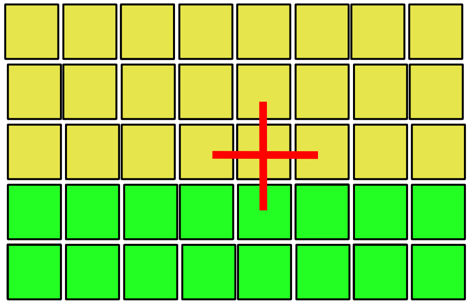
Access Granularity: Banded Stencil

Communication traffic:
2 * N * THREADS elements
2 * N * THREADS accesses

Better solution: banded layout

```
shared [N*N/THREADS] double A[N][N];

upc_forall (i=1; i<N-1; i++; continue)
  upc_forall (j=1; j<N-1; j++; &A[i][j])
    A[i][j] = 0.25 *(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1];
```

Better, because only 2*N accesses per thread are non-local



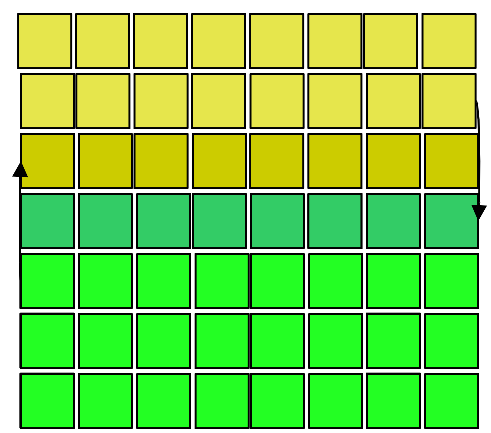
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Access Granularity: Shadow Exchange

Communication traffic:
2 * N * THREADS elements
2 * THREADS accesses

Banded layout with shadow regions:

```
#define B (N*(N+2*THREADS)/THREADS)
shared [B] double A[N+2*THREADS][N];

/* exchange shadows (code incomplete, no bounds checks!) */
int l=MYTHREAD*B;              /* lower shadow */
upc_memget (&A[l][0], &A[l-2][0], N*sizeof(double));
int u=(MYTHREAD+1)*B-1;        /* upper shadow row */
upc_memget (&A[u][0], &A[u+2][0], N*sizeof(double));

/* stencil code as usual */
...
```



Shadow region exchange

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

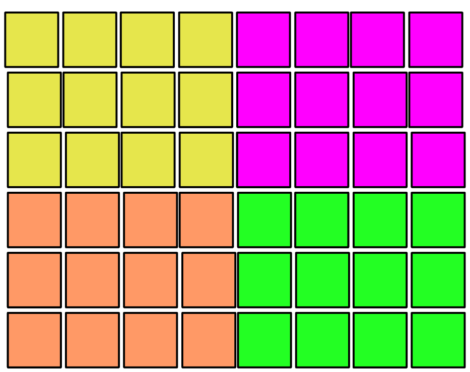
Access Granularity: Tiled layout

Communication traffic:
4 * N * sqrt(T) elements
4 * T accesses

Tiled layout (UPC extension)

```
#define B
shared [B][B] double A[N][N];
```

- **Very complicated code (exchange buffers are not contiguous) (*)**
- **Highly scalable: per-thread communication costs <u>decrease</u> with scaling**



(*) compiler aggregation optimization can help keep code small

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Matrix multiplication: Introduction

```
shared double A[M][P], B[P][N], C[M][N];

forall (i=0; i<M; i++; continue)
  forall (j=0; j<N; j++; &C[i][j])
     for (k=0; k<P; k++)
        C[i][j] += A[i][k]*B[k][j];
```

Problem:

- **Accesses to A and B are mostly non-local**
- **Fine grain remote access == bad performance!**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

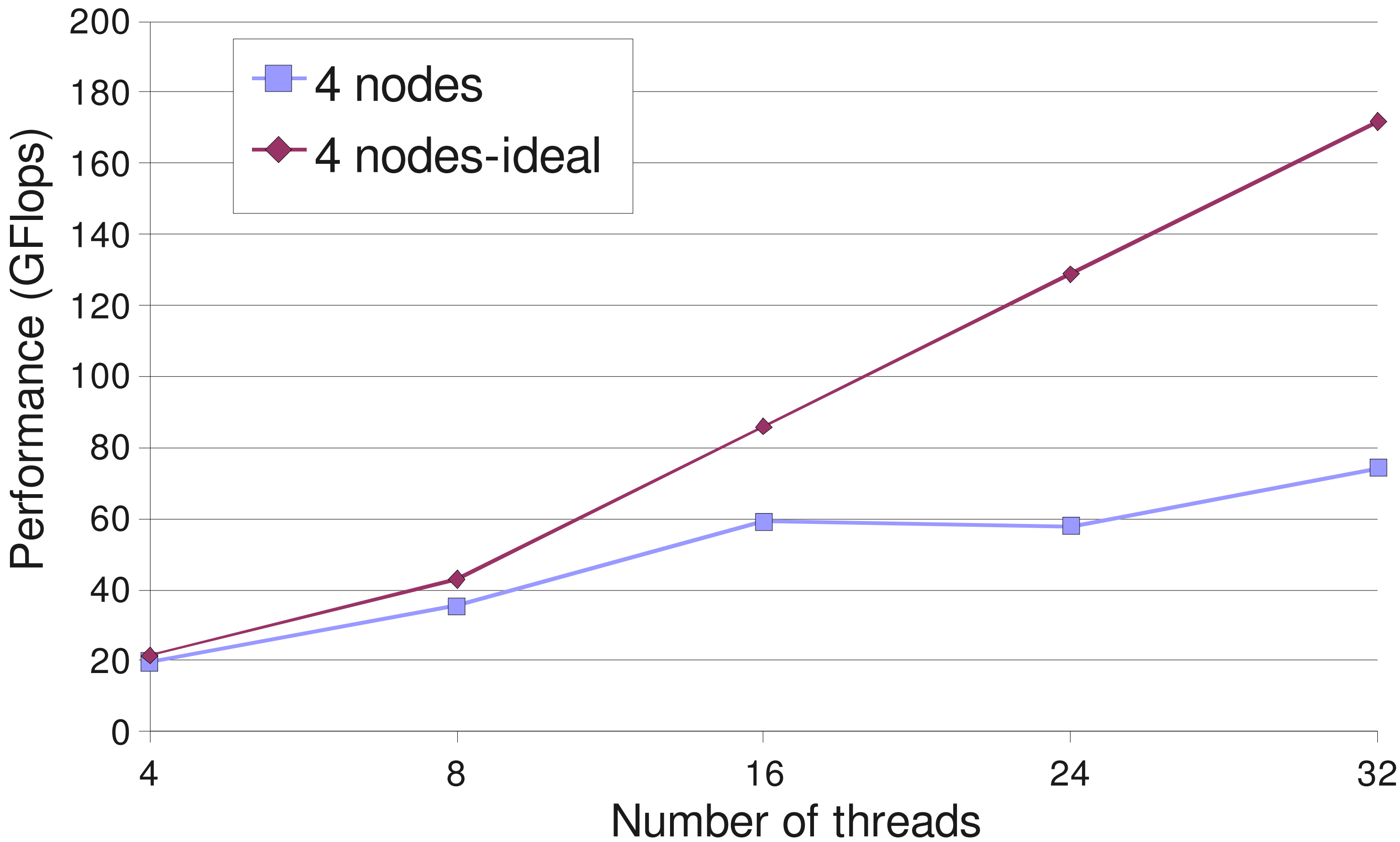
Matrix multiplication: Block matrices

```
shared struct { int x[B][B]; } A[M1][P1], B[P1][N1], C[M1]
  [N1];

forall (i=0; i<M1; i++; continue)
  forall (j=0; j<N1; j++; &C[i][j])
     for (k=0; k<P1; k++) {
        upc_memget (alocal, &A[i][k], B*B*sizeof(double));
        upc_memget (blocal, &B[k][j], B*B*sizeof(double));
        dgemm (alocal, blocal, &C[i][j]);
     }
```

– **Good:**
  - **Fewer accesses, large granularity**
  - **Improved single-node performance (ESSL library call)**
– **Bad:**
  - **Code has changed significantly**
  - **Still not scalable performance: O(n^3) communication**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Blocked Matrix Multiply scaling
P5 cluster, 4 nodes x 8 threads/node



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

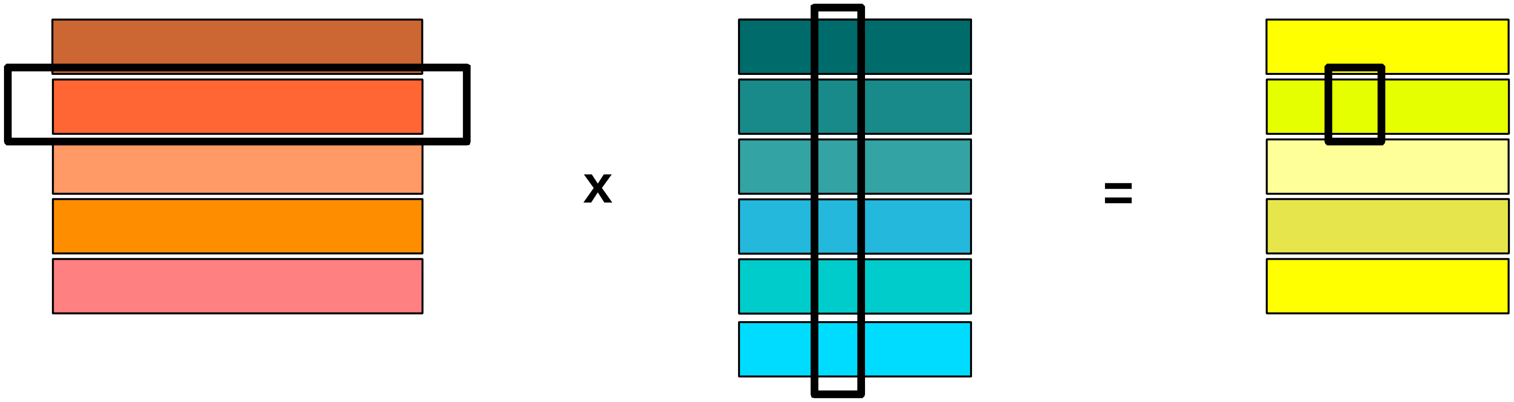
Matrix multiplication: New Layout

```
typedef shared { int x[B][B]; } Block;
shared [M1*sizeof(Block)] Block A[M1][P1];
shared [P1*sizeof(Block)] Block B[P1][N1];
shared [M1*sizeof(Block)] Block C[M1][N1];
```

Good:

- no locality issues on A and C (traversing the same way)

- Bad:
  - B is traversed across block layout (communication!)

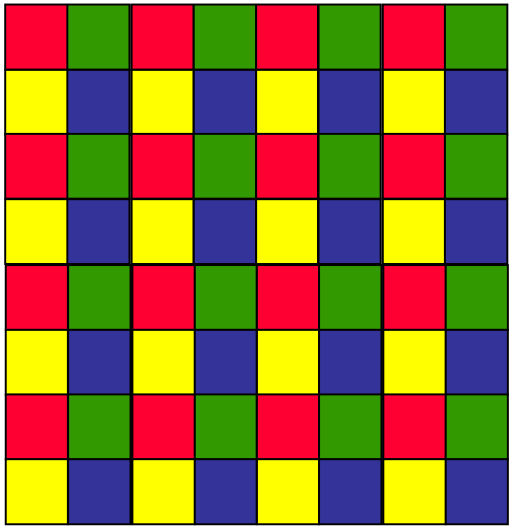


This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Matrix Multiplication: Tiled Layouts

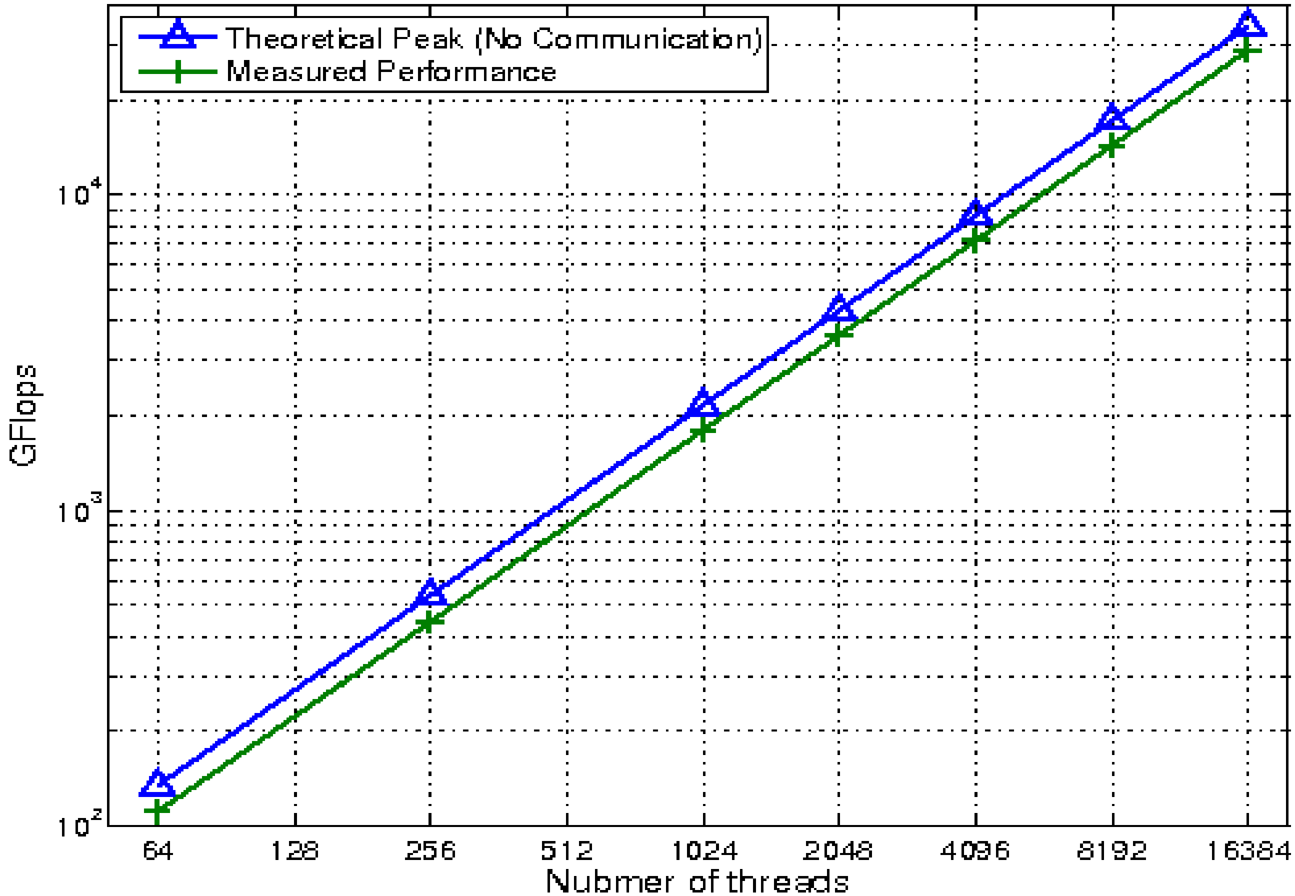
```
#pragma processors C(Tx, Ty)
shared [B][B] double C[M][N];
```

- **Good:**
  - **Allows control of block placement on processor grid**
  - **Allows C to be accessed as array, not as struct**
  - **Allows communication among rows, cols of processors (scalable communication)**
- **Bad:**
  - **UPC extension: not available in vanilla UPC**
  - **Not <u>yet</u> available in IBM UPC**
- **Good:**
  - **Attempting to add this into standard**



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Scalability: Matrix multiplication: Tiled layout key to performance



UPC matrix multiplication on a 16-rack Blue Gene/L

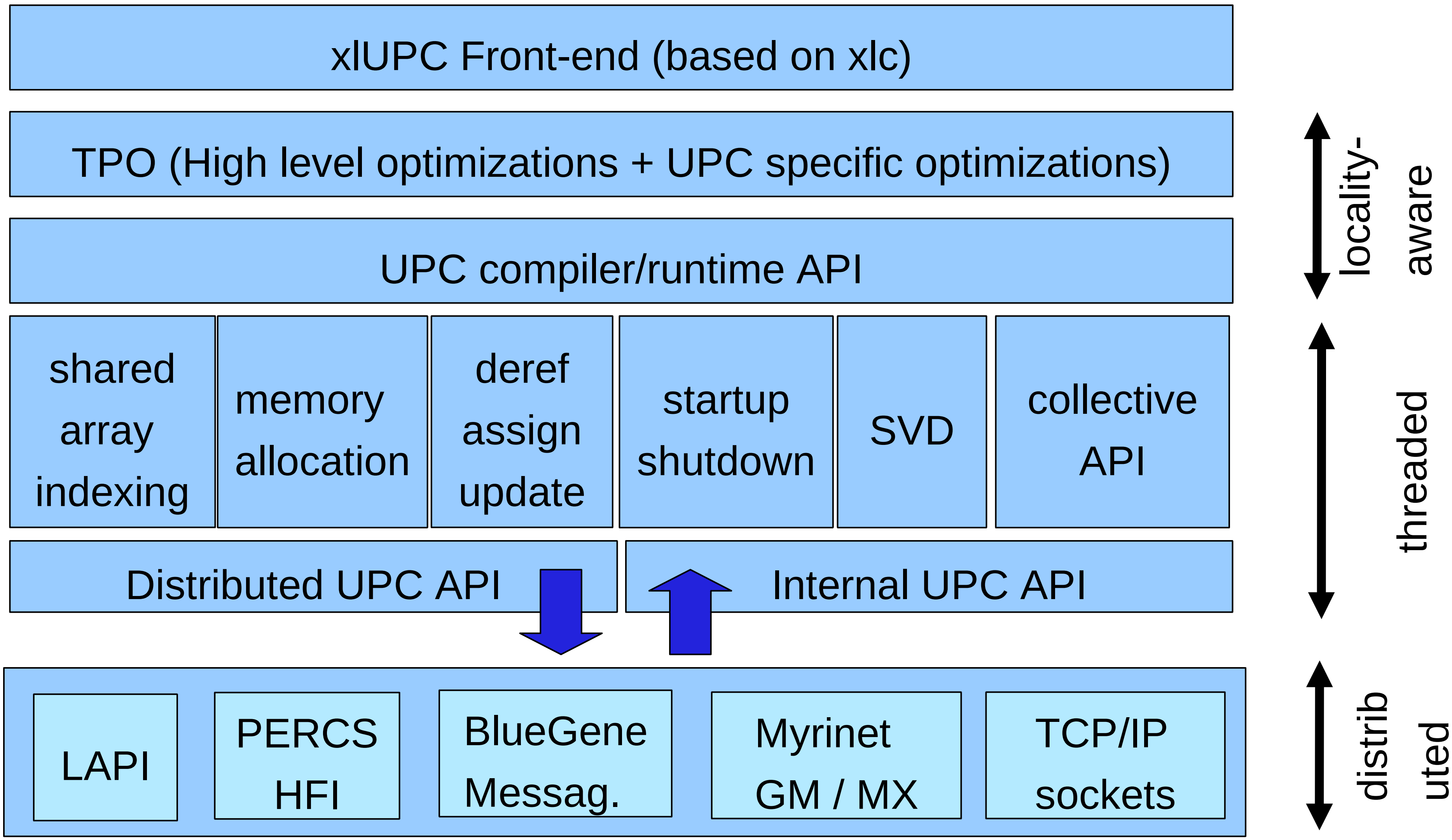
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

3. Compiler optimizations

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

IBM

UPC hybrid runtime stack



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

XL-UPC Runtime System

- **Designed for scalability**
- **Implementations available for**
  - SMP using pthreads
  - Clusters using LAPI
  - BlueGene/L using the BG/L message layer
- **Provides a unique API to the compiler for all the above implementations**
- **Provides management of and access to shared data in a scalable manner using the Shared Variable Directory**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

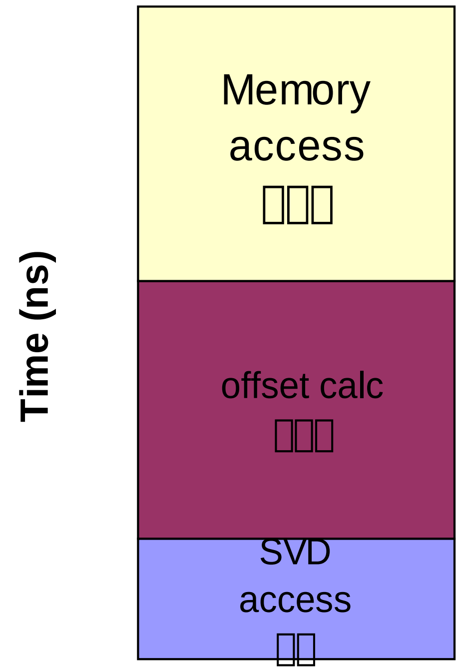
Anatomy of a shared access

```
shared [BF] int A[N],B[N],C[N];
upc_forall (i=0; i < N, ++i; &A[i])
    A[i] = B[i] + C[i];
```

Generated code (loop body):

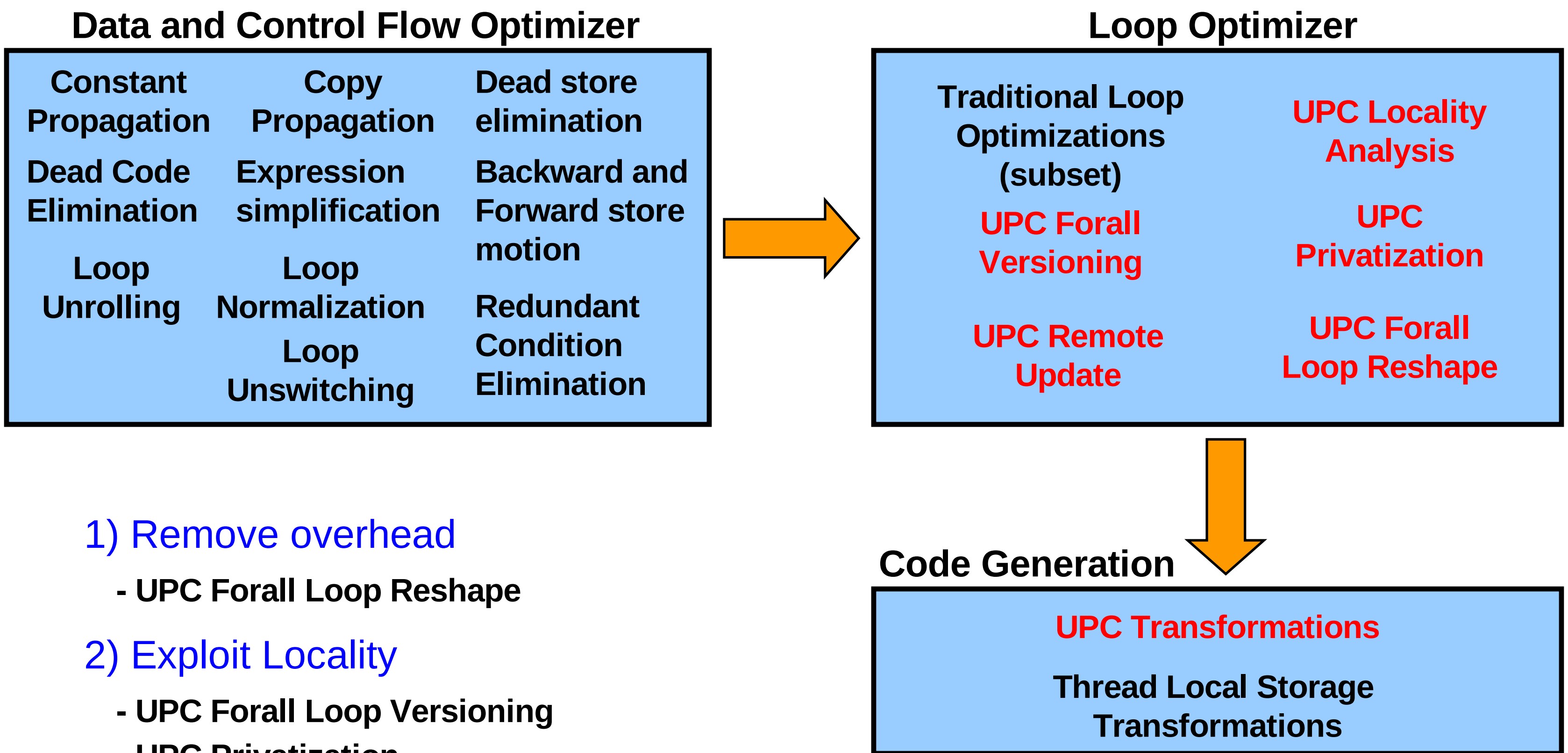
```
__xlupc_deref_array(C_h, __t1, i, sizeof(int), …);
__xlupc_deref_array(B_h, __t2, i, sizeof(int), …);
__t3 = __t1 + __t2;
__xlupc_assign_array(A_h, __t3, i, sizeof(int), …);
```

Anatomy of a runtime call



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Optimizer Infrastructure



1) Remove overhead
   - **UPC Forall Loop Reshape**

2) Exploit Locality
   - **UPC Forall Loop Versioning**
   - **UPC Privatization**

3) Reduce communication
   - **UPC Remote Update**

Optimizer infrastructure applicable to other PGAS languages (Co-Array Fortran)

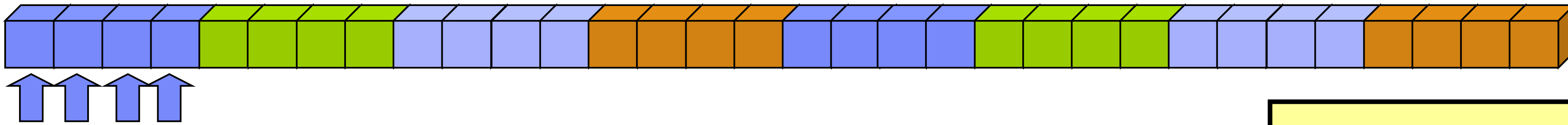
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect

Optimizing Shared Object Accesses

- **When shared data is accessed, function calls to the RTS are used to set or get the shared data**
- **The calls to the RTS use the Shared Variable Directory (SVD) to locate the shared data**
  - This lookup requires several pointer dereferences and is expensive
- **When the compiler can determine the *relative* thread that owns each shared reference, it can perform several different optimizations to reduce the overhead of RTS calls**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Locality Analysis



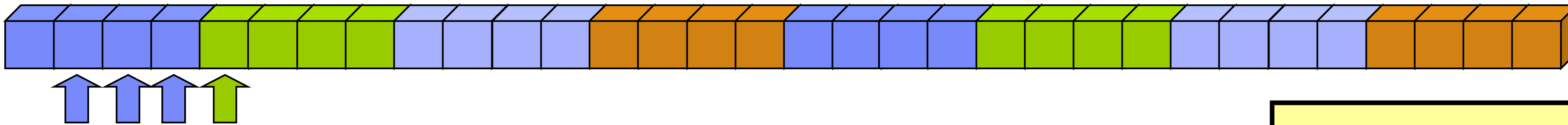
4 UPC Threads

```
shared [4] int A[Z][Z], B[Z][Z], C[Z][Z];
int main ( ) {
   int k , l ;
   for (k =0; k<Z ; k++) {
         upc_forall (l=0; l < Z; l++;  &A[k][l]) {
               A[k][l] = 0 ;
               B[k][l+1] = m+2;
               C[k][l+14] = m*3;
         }
   }
}
```

All accesses of A[k][l] are local to executing thread

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Locality Analysis



```
shared [4] int A[Z][Z], B[Z][Z], C[Z][Z];
int main ( ) {
   int k , l ;
   for (k =0; k<Z ; k++) {
         upc_forall (l=0; l < Z; l++;  &A[k][l]) {
               A[k][l] = 0 ;
               B[k][l+1] = m+2;
               C[k][l+14] = m*3;
         }
   }
}
```

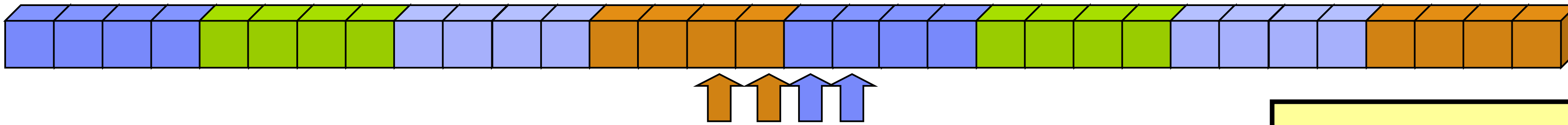
4 UPC Threads

Some accesses of A[k][l] are local, some are remote

The point where the locality changes is called the ***cut***

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Locality Analysis



4 UPC Threads

```
shared [4] int A[Z][Z], B[Z][Z], C[Z][Z];
int main ( ) {
  int k , l ;
  for (k =0; k<Z ; k++) {
      upc_forall (l=0; l < Z; l++;  &A[k][l]) {
          A[k][l] = 0 ;
          B[k][l+1] = m+2;
          C[k][l+14] = m*3;
      }
  }
}
```

Some accesses of A[k][l] are remote, some are local

Account for the block-cyclic distribution in UPC

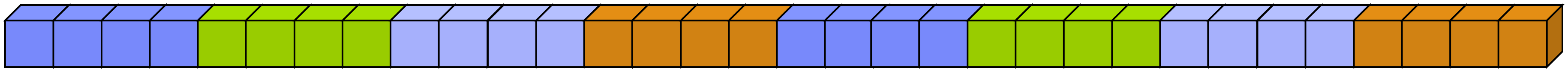
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Locality Analysis

- **Compiler refactors the original loop nest into regions where the locality is constant**
- **The regions are created using *cuts***
- **Once the loop is refactored, the compiler builds a *Shared Reference Map* that maps the shared references to the relative thread that owns them**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Shared Reference Map



```
shared [4] int A[N][N], B[N][N], C[N][N] ;
int main ( ) {
  int i, j ;
  for (i=0; i < N; i++) {
    upc_forall (j=0; j < N; j++; &A[i][j])  {
      if (j < 2) {
        A[i][j] = 0 ; / / A1
        B[i][j+1] = m+2; / / B1
        C[i][j+14] = m*3; / / C1
      } else if (j < 3) {
        A[i][j] = 0 ; / / A2
        B[i][j+1] = m+2; / / B2
        C[i][j+14] = m*3; / / C2
      } else {
        A[i][j] = 0 ; / / A3
        B[i][j+1] = m+2; / / B3
        C[i][j+14] = m*3; / / C3
      }
```

Region 2
Position [0,0]

Region 3
Position [0,2]

Region 4
Position [0,3]

Thread		Shared References
MYTHREAD	➡	A1 B1 A2 B2 C2 A3 C3
MYTHREAD+1	➡	B3
MYTHREAD+2	➡	
MYTHREAD+3	➡	C1

HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect

Shared Reference Map

- **The shared reference map gives the compiler information about the relative thread that owns each shared reference**
- **The shared reference map is not concerned about machine configuration**
- **This information can then be used to perform locality optimizations**
  - **Privatization**: Local shared references can be accessed directly without requiring the RTS
  - **Coalescing**: Remote shared references owned by the same thread can be accessed in groups
  - **Scheduling**: Remote shared references that require communication

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Shared Object Access Privatization

- **All shared references that map to MYTHREAD in the SRM are *local* to the accessing thread**
- **Machine architecture can also be used to find accesses for co-located threads**
- **Compiler converts each local shared-object access into a traditional C pointer access**
  - Base address obtained from RTS ***once***
  - Offset computed based on index and shared array shape

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAP Example: Stream Triad

```
#define SCALAR 3.0
shared double a[N], b[N], c[N];
void StreamTriad() {
  int i;
  upc_forall(i=0;i<N;i++;&a[i])
    a[i] = b[i] + SCALAR*c[i];
}
```

```
#define SCALAR 3.0
shared double a[N], b[N], c[N];
void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; i) {
    __xlupc_deref_array(c_h, tmp1, i);
    __xlupc_deref_array(b_h, tmp2, i);
    tmp3 = tmp2 + 3.0*tmp1;
    __xlupc_assign_array(a_h, tmp3, i);
  }
}
```

Naïve transformation results in 3 calls to the UPC Runtime in every iteration

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAP Example: Stream Triad

Get base addresses outside of loop

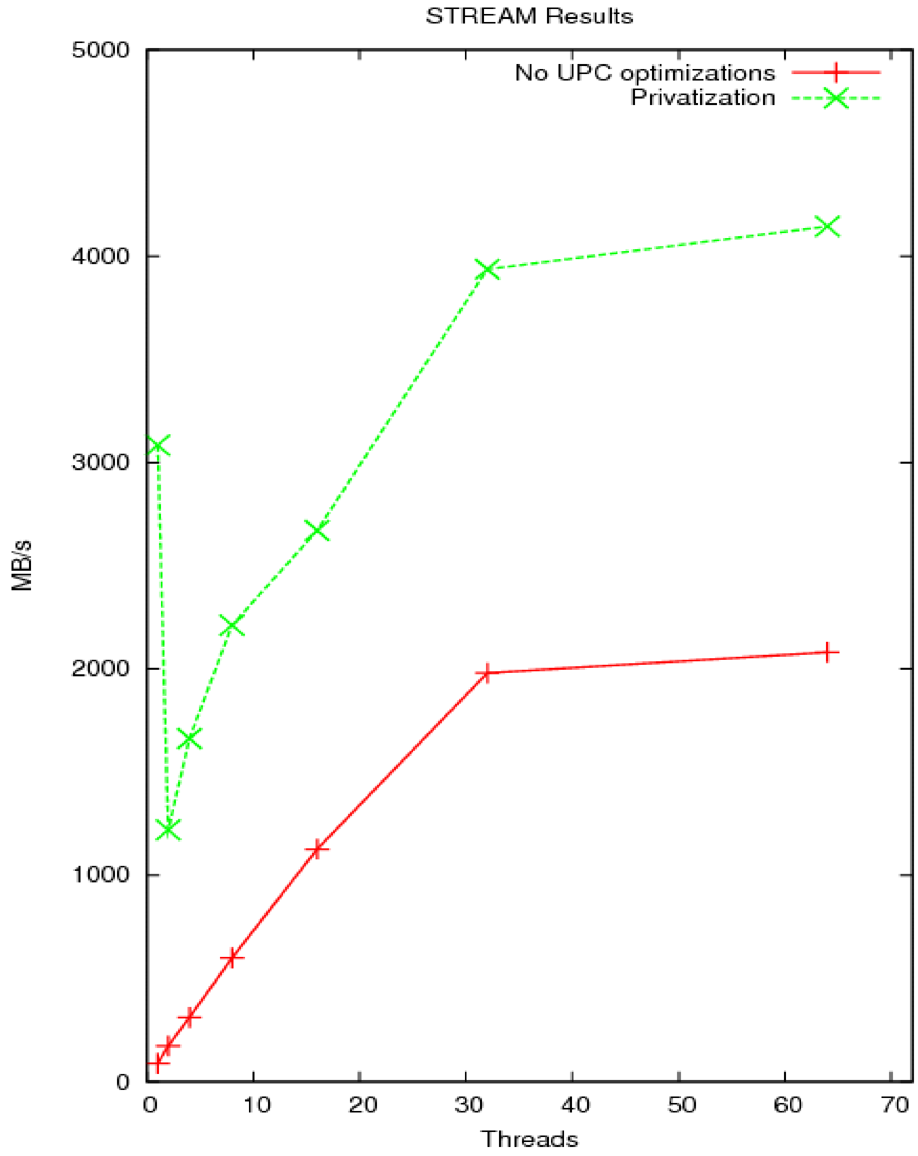
```
#define SCALAR 3.0
shared double a[N], b[N], c[N];
void StreamTriad() {
  int i;
  upc_forall(i=0;i<N;i++;&a[i])
    a[i] = b[i] + SCALAR*c[i];
}
```

```
#define SCALAR 3.0
shared double a[N], b[N], c[N];
void StreamTriad() {
  int i;
  aBase = __xlupc_base_address(a_h);
  bBase = __xlupc_base_address(b_h);
  cBase = __xlupc_base_address(c_h);
  upc_forall(i=0;i<N;i++;&a[i]) {
    aOffset = ComputeOffset(i);
    bOffset = ComputeOffset(i);
    cOffset = ComputeOffset(i);
    *(aBase+aOffset) = *(bBase+bOffset) +
                  SCALAR*( *(cBase + cOffset));
  }
}
```

Traditional “C” pointer access

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAP Example: Stream Triad



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Shared Object Access Coalescing

- **All shared references that map to the same remote thread can be *coalesced* together**
- **This reduces the number of messages, thereby reducing the execution time**
- **Requires support from the UPC Runtime**
- **Current runtime interface requires coalesced shared references to have**
  - The same base symbol
  - The same owner
  - The same type (read or write)

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAC Example: Sobel Edge Detection

```
shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
int Sobel() {
  int i, j, gx, gy;
  double gradient;
  for (i=1; i < ROWS-1; i++) {
    upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
      gx =   (int) orig[i-1][j+1] - orig[i-1][j-1];
      gx += ((int) orig[i][j+1]   - orig[i][j-1]) * 2;
      gx +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
      gy =   (int) orig[i+1][j-1] - orig[i-1][j-1];
      gy += ((int) orig[i+1][j]   - orig[i-1][j]) * 2;
      gy +=  (int) orig[i+1][j+1] - orig[i-1][j+1];
      gradient = sqrt((gx*gx) + (gy*gy));
      if (gradient > 255) gradient = 255;
      edge[i][j] = (BYTE) gradient;
    } } }
```

Shared arrays blocked by row

Loop body contains 12 shared array access

3 Local Accesses

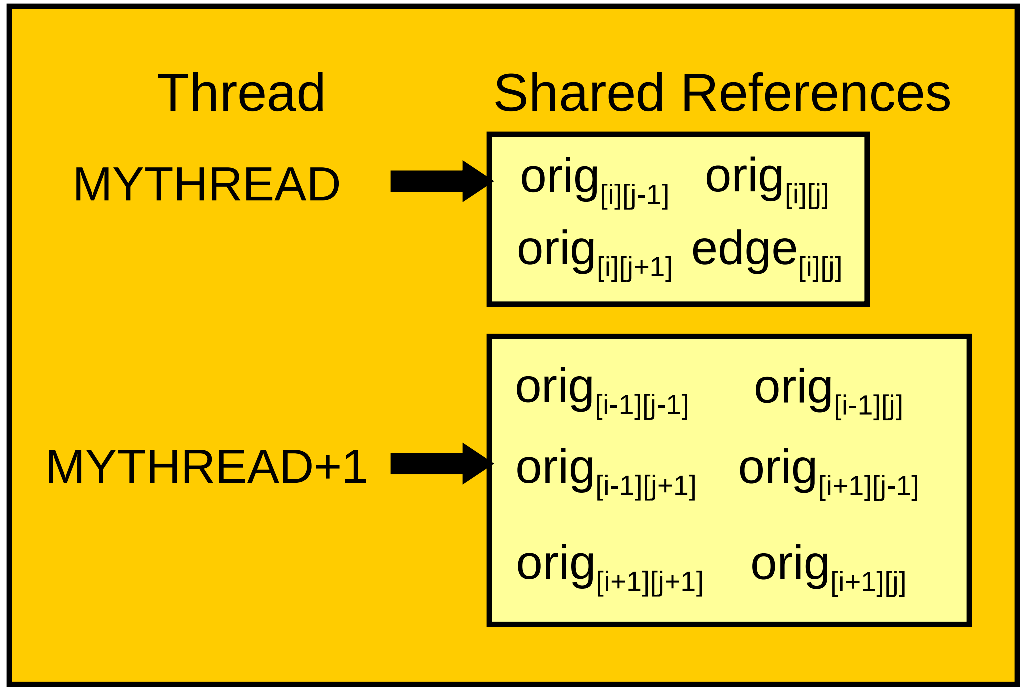
9 Remote Accesses

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAC Example: Sobel Edge Detection

2 UPC Threads

```
shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
int Sobel() {
  int i, j, gx, gy;
  double gradient;
  for (i=1; i < ROWS-1; i++) {
    upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
      gx =   (int) orig[i-1][j+1] - orig[i-1][j-1];
      gx += ((int) orig[i][j+1]   - orig[i][j-1]) * 2;
      gx +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
      gy =   (int) orig[i+1][j-1] - orig[i-1][j-1];
      gy += ((int) orig[i+1][j]   - orig[i-1][j]) * 2;
      gy +=  (int) orig[i+1][j+1] - orig[i-1][j+1];
      gradient = sqrt((gx*gx) + (gy*gy));
      if (gradient > 255) gradient = 255;
      edge[i][j] = (BYTE) gradient;
  } } }
```



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

SOAC Example: Sobel Edge Detection

2 UPC Threads

```
shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
int Sobel() {
  int i, j, gx, gy;
  double gradient;
  for (i=1; i < ROWS-1; i++) {
    upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
      CoalescedRemoteAccess(tmp1, orig, ((i-1)*COLUMNS)+j-1, 3, 1);
      CoalescedRemoteAccess(tmp2, orig, ((i+1)*COLUMNS)+j-1,3, 1);
      gx = (int) tmp1[2] - tmp1[0];   gx += ((int) orig[i][j+1]   - orig[i][j-1]) * 2;
      gx += (int) tmp2[2] - tmp2[0];  gy = (int) tmp2[0] - tmp1[0];
      gy += ((int) tmp2[1] - tmp1[1] * 2;  gy += (int) tmp2[2] - tmp1[2];
      gradient = sqrt((gx*gx) + (gy*gy));
      if (gradient > 255) gradient = 255;
      edge[i][j] = (BYTE) gradient;
  } } }
```

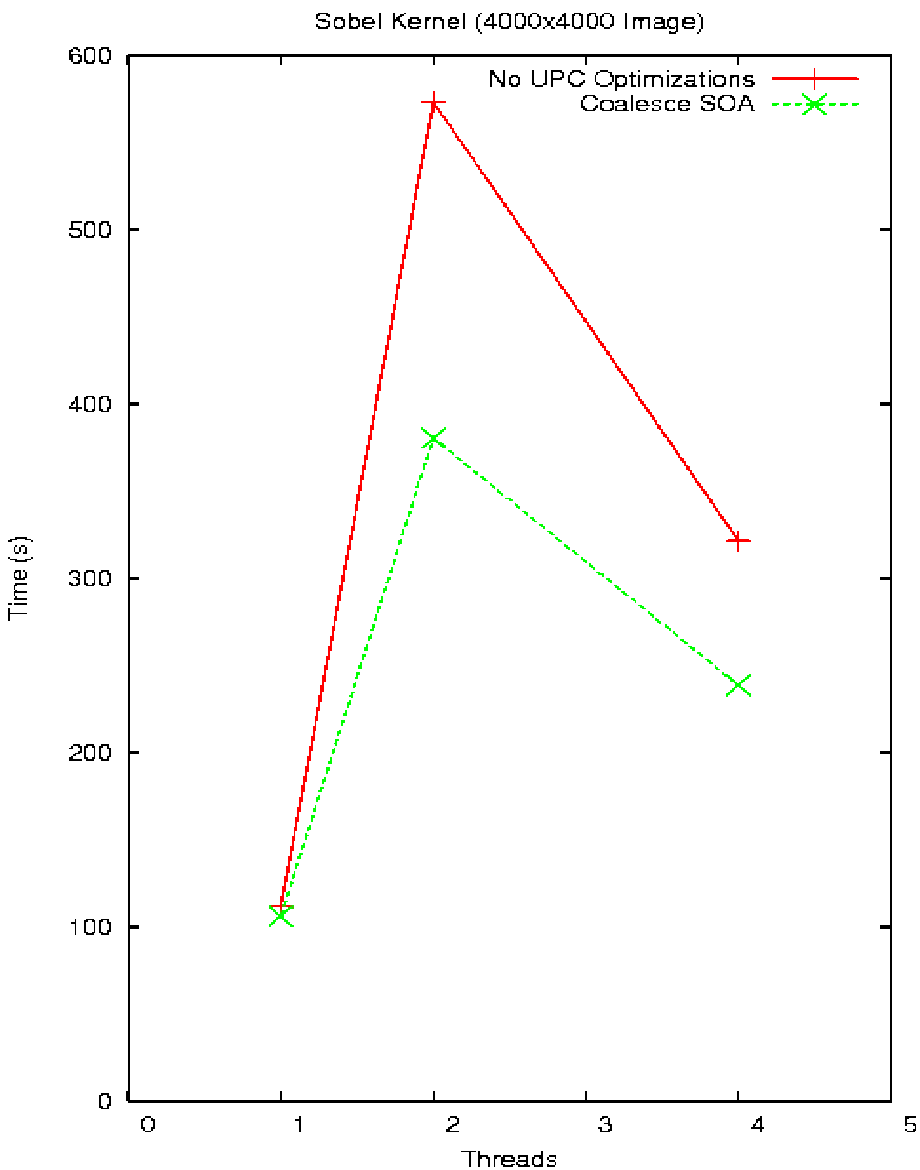
Two calls to the UPC runtime to retrieve remote shared data

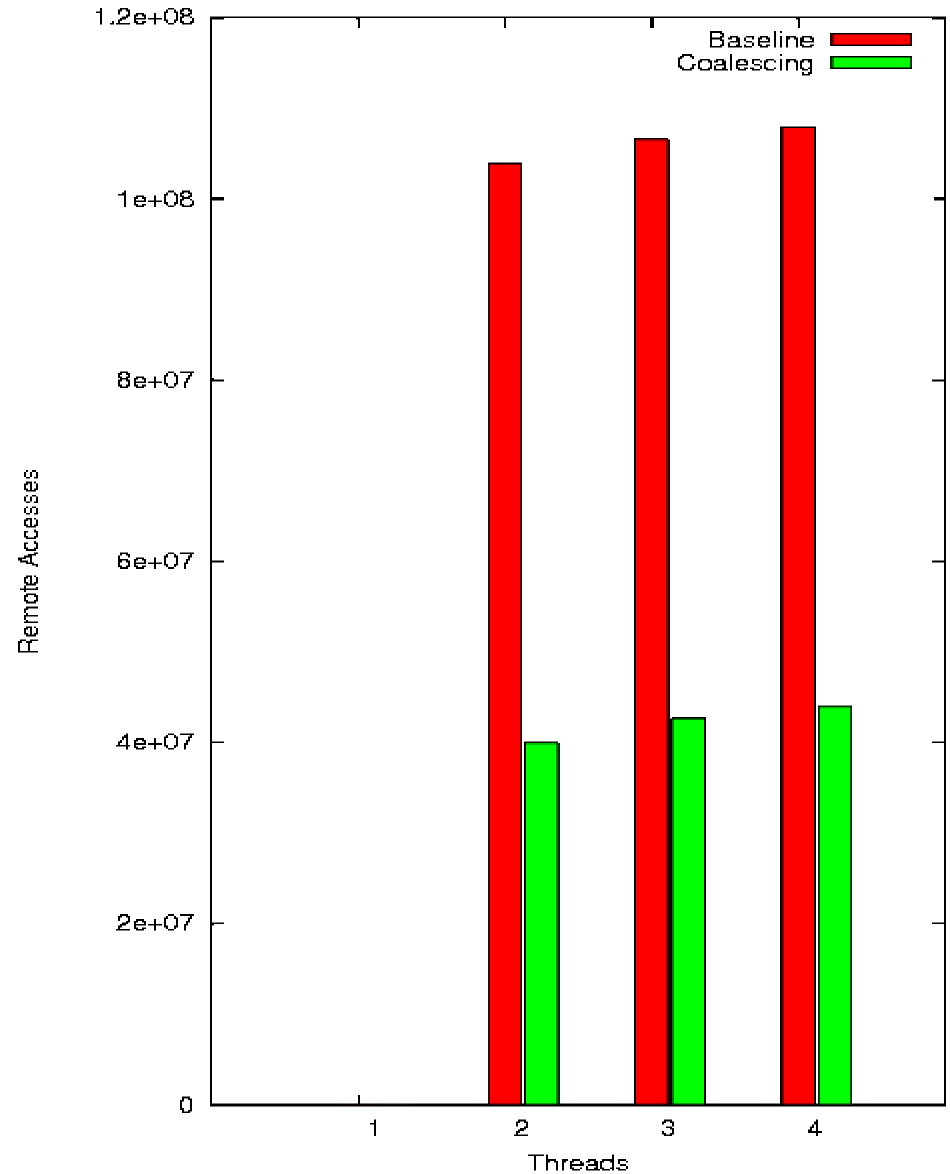
Temporary buffers created and managed by compiler

All accesses are now local

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAC Example: Sobel Edge Detection





This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Shared-Object Access Scheduling

- In UPC when a thread executes a runtime call it will wait until the call has been serviced.

```
__xlupc_deref_array(C_h, __t1, i, sizeof(int), …);     ← wait
__xlupc_deref_array(B_h, __t2, i, sizeof(int), …);     ← wait again
__t3 = __t1 + __t2;
__xlupc_assign_array(A_h, __t3, i, sizeof(int), …);  ← wait again !
```

- Runtime calls are blocking, but we could use non-blocking pairs of calls (notify/wait)

```
req = __xlupc_deref_array_elt_post(…);
__xlupc_wait(req);
```

- Notify call initiates the RT access of the shared object
- Wait call should be placed before the first use

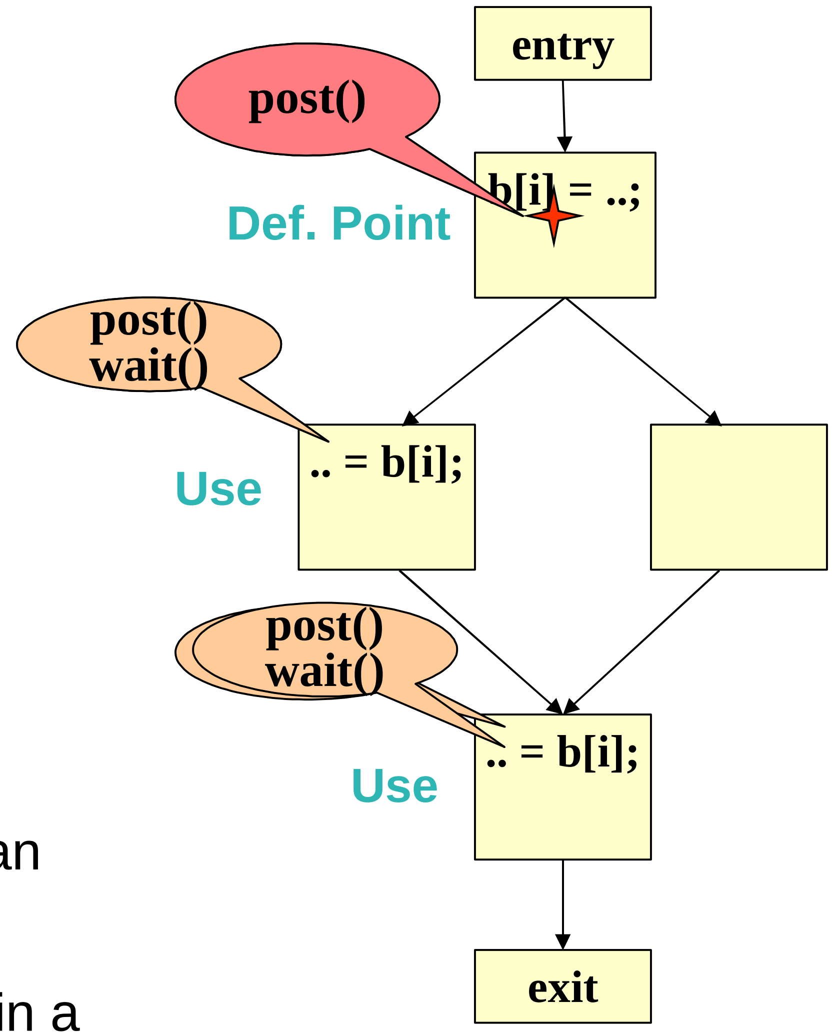
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Scheduling of post/wait calls

Work in Progress

GOAL: place the post call as early as possible

- Collect shared loads used in a loop
- Split the blocking calls in a post/wait pair
- Find the definition point of the shared ref.
- *Move the post call to the earliest program point with the following properties:*
  - *shared ref. is executed on all paths from P to exit*
  - *P is dominated by its shared ref. definition*
- Redundant post calls for the same object can be eliminated
- Consecutive post calls may be aggregated in a bulk transfer



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAS Example: Synthetic Benchmark

```
shared double MySharedData[SDS], RemoteData[SDS];
double MyPrivateData[PDS];
void Compute() {
  int i, j, k;
  double sum=0.0;
  upc_forall (i=0; i < SHARED_DATA_SIZE; i++; &MySharedData[i]) {
    for (j=0; j < PRIVATE_DATA_SIZE; j++) {
      sum += MyPrivateData[j];
    }
    MySharedData[i] = sum * RemoteData[(i+1)%(SDS)];
  }
}
```

Local Computation

Remote access

Execution waits until access is finished

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

SOAS Example: Synthetic Benchmark

```
shared double MySharedData[SDS], RemoteData[SDS];
double MyPrivateData[PDS];
void Compute() {
 int i, j, k;
 double sum=0.0;
 wait = deref_array_post(RemoteData_h, MYTHREAD);
 upc_forall (i=0; i < SHARED_DATA_SIZE; i++; &MySharedData[i]) {
  for (j=0; j < PRIVATE_DATA_SIZE; j++) {
   sum += MyPrivateData[j];
  }
  offset=THREADS;
  deref_array_wait(wait);
  tmp2 = tmp;
  if (i+THREADS) < SHARED_DATA_SIZE)
   wait = deref_array_post(RemoteData_h,&tmp,(i+THREADS+1)%SHARED_DATA_SIZE);
  MySharedData[i] = sum * tmp2;
 } }
```

Prefetch first iteration

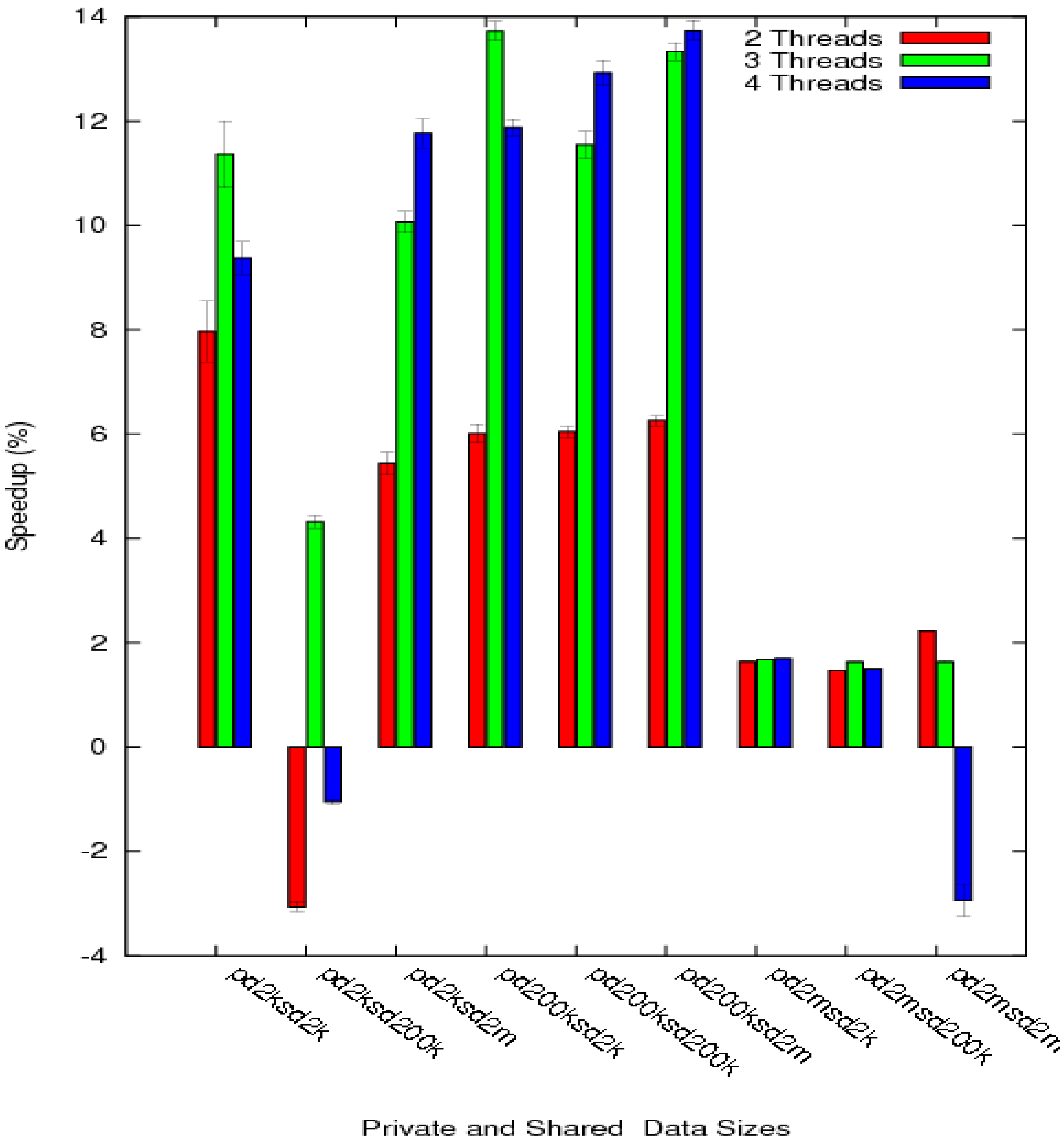
Local Computation

Wait for data from previous iteration

Prefetch next iteration

HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect

SOAC Example: Synthetic Benchmark



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

The upc_forall work-sharing construct

- Similar to C for loop, 4th field indicates the affinity
  - *Integer affinity:* Thread with ID == affinity value executes the iteration
  - *Pointer-to-shared affinity:* thread that "owns" shared element executes the iteration
- Affinity test exec. on *each* iteration by all threads

Affinity Test

```
shared int A[N],B[N],C[N];
upc_forall(i=0; i < N; i++; i){
    A[i] = B[i] + C[i];
}
```

Branch

```
shared int A[N],B[N],C[N];
for (i=0; i < N; i++)  {
  if ((i % THREADS)== MYTHREAD)
     A[i] = B[i] + C[i];
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.
HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

The upc_forall work-sharing construct

- Similar to C for loop, 4th field indicates the affinity
  - *Integer affinity:* Thread with ID == affinity value executes the iteration
  - *Pointer-to-shared affinity:* thread that "owns" shared element executes the iteration
- Affinity test exec. on *each* iteration by all threads

Affinity Test

```
shared [BF] int A[N],B[N],C[N];
upc_forall(i=0; i < N; i++; &A[i]) {
    A[i] = B[i] + C[i];
}
```

Branch

```
shared [BF] int A[N],B[N],C[N];
for (i=0; i < N; i++) {
  if (upc_threadof(A[i]) == MYTHREAD)
     A[i] = B[i] + C[i];
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Integer Affinity)

- **Iteration space is partitioned**
- **Each thread starts executing at iteration *MYTHREAD***
- **Each thread executes every *THREADs* elements**

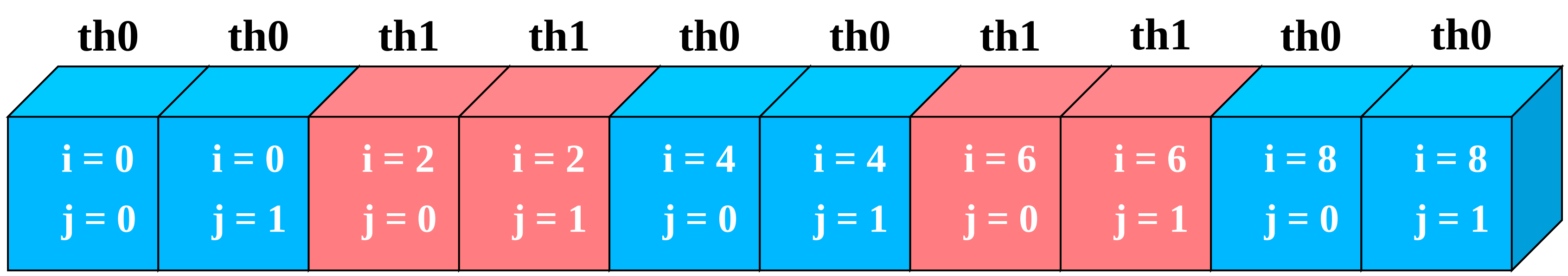
```
shared int A[N],B[N],C[N];
upc_forall(i=0; i < N; i++; i){
    A[i] = B[i] + C[i];
}
```

```
shared int A[N], B[N], C[N];
for (i=MYTHREAD; i < N; i+= THREADS) {
    A[i] = B[i]+C[i];
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

- **Strip-mining optimization**
- **Create a 2-level loop nest**
  - Outer loop iterates over blocks owned by MYTHREAD
  - Inner loop iterates through each block element




```
shared [2] int A[N], B[N], C[N];
for (i=MYTHREAD * 2; i < N; i+= THREADS*2) {
    for (j=i; j < i+BF; j++)
        A[j] = B[i]+C[i];
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

- **Consider an upper-triangle parallel loop nest that uses pointer-to-shared affinity**
- **Shared array elements are initialized to -1**
- **Shared array elements are assigned the thread ID of their owner**

```
shared [ 2 ] double A[ 6 ] [ 6 ] ;

void UpperTriangularLoop ( ) {
  int  i , j ;

  for (i =0; i < 6 ; i ++)

    upc_forall (j=i; j<6; j++; &A[i][j])

      A[ i ] [ j ] = ( double ) MYTHREAD;
}
```

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

- **For every iteration of the i loop, the upc_forall loop has a different iteration vector**
- **Thus, the compiler must compute the bounds of the new loop nest at runtime**

```
shared [ 2 ] double A[ 6 ] [ 6 ] ;

void UpperTriangularLoop ( ) {
  int  i , j ;

  for (i =0; i < 6 ; i ++)

      upc_forall (j=i; j<6; j++; &A[i][j])

           A[i] [j] = (double) MYTHREAD;
}
```

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

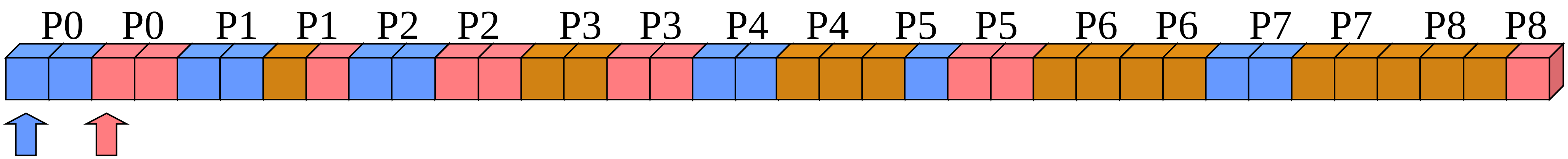
- **The compute bound functions determine the iteraton vector for the new loop nests at runtime**
  - For each iteration of the i loop, compute the position of the next block owned by thread MYTHREAD

```
shared [ 2 ] double A[ 6 ] [ 6 ] ;

void UpperTriangularLoop() {
    int i, j;
    outerLB  = ComputeOuterLowerBound();
    outerUB = ComputeOuterUpperBound();
    innerLB  = ComputeInnerLowerBound();
    innerUB = ComputeInnerUpperBound();

    for (i=0; i<6 ;i++)
        for  (k=outerLB; k<outerUB; k++)
            for (l=innerLB; l < innerUB; l++)
                A[i] [k+l] = (double) MYTHREAD;
}
```

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

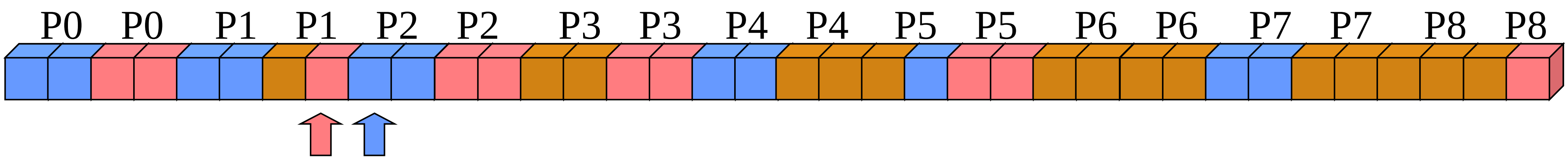


	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1		
i=2		
i=3		
i=4		
i=5		

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

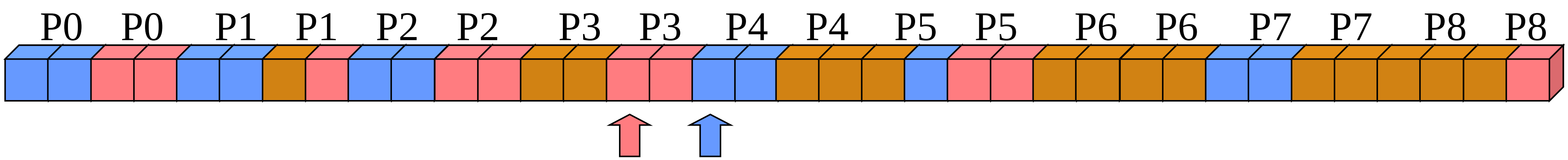


	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1	P=2, C=0	P=1, C=1
i=2		
i=3		
i=4		
i=5		

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect

Parallel Loop Reshaping (Pointer-to-shared affinity)

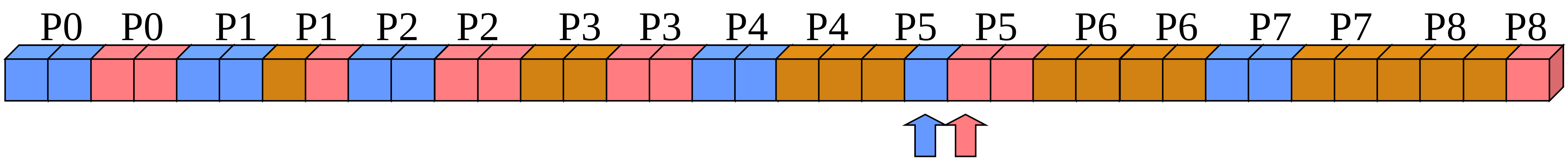


	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1	P=2, C=0	P=1, C=1
i=2	P=4, C=0	P=3, C=0
i=3		
i=4		
i=5		

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

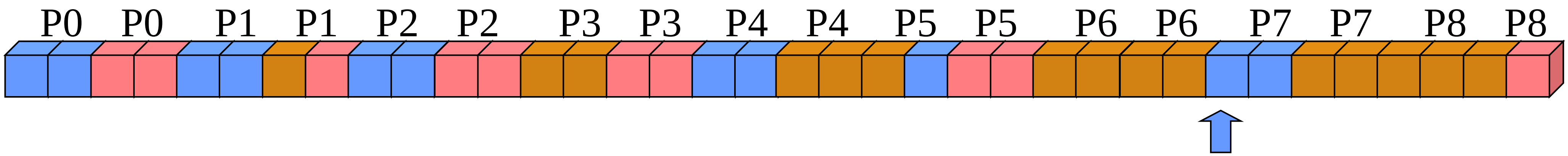


	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1	P=2, C=0	P=1, C=1
i=2	P=4, C=0	P=3, C=0
i=3	P=5, C=1	P=5, C=0
i=4		
i=5		

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)

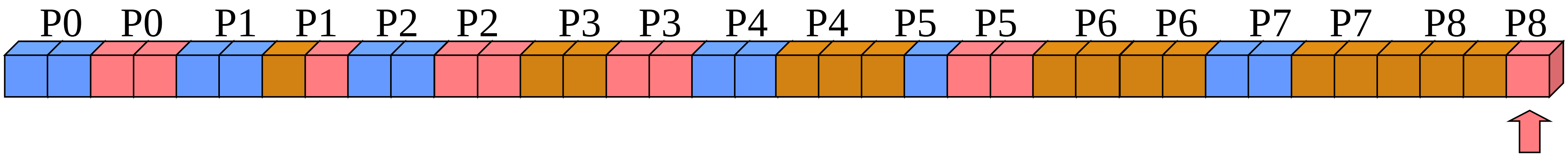


	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1	P=2, C=0	P=1, C=1
i=2	P=4, C=0	P=3, C=0
i=3	P=5, C=1	P=5, C=0
i=4	P=7, C=0	P=0, C=0
i=5		

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping (Pointer-to-shared affinity)



	Thread 0	Thread 1
i=0	P=0, C=0	P=0, C=0
i=1	P=2, C=0	P=1, C=1
i=2	P=4, C=0	P=3, C=0
i=3	P=5, C=1	P=5, C=0
i=4	P=7, C=0	P=0, C=0
i=5	P=0, C=0	P=8, C=1

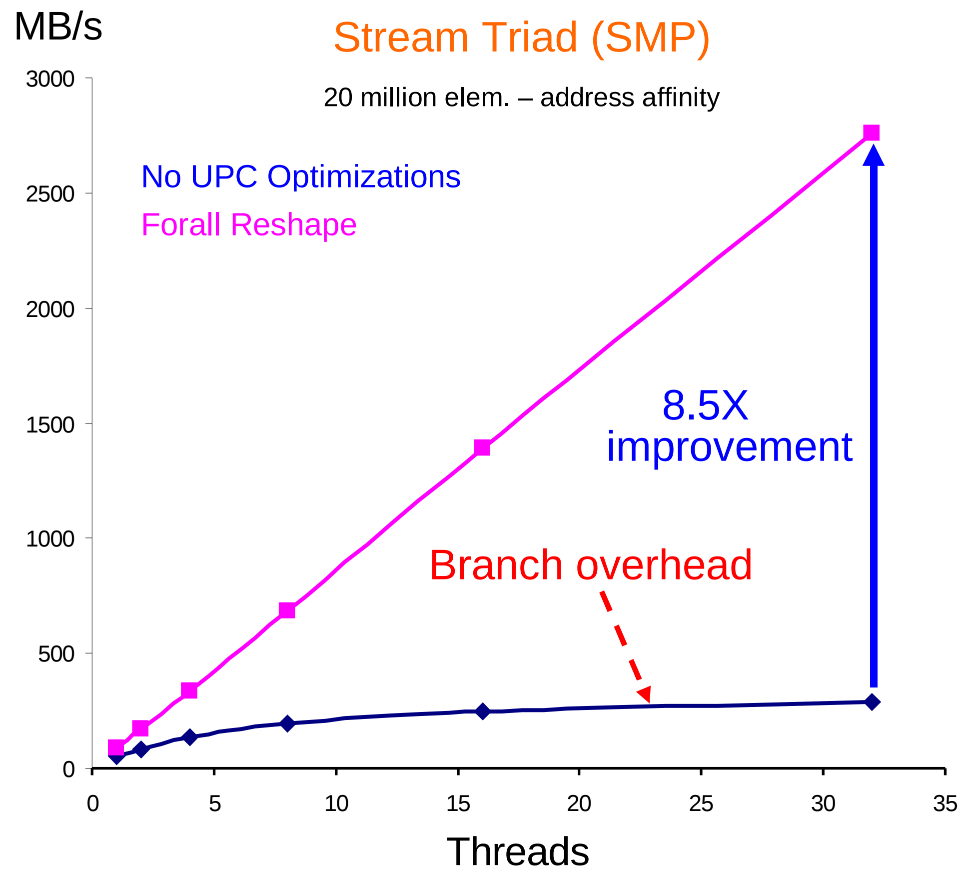
0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping

Removed overhead

Objective: **remove branch overhead in naïve upc_forall translation**



AIX 5.3, Power5, 2.3 GHz

```
shared [BF] int A[N],B[N],C[N];
upc_forall (i=0; i < N; i++; &A[i])
    A[i] = B[i] + k*C[i];
```

naïve translation

```
shared [BF] int A[N],B[N],C[N];
for (i=0; i < N; i++)
  if (upc_threadof(A[i]) == MYTHREAD)
    A[i] = B[i] + k*C[i];
```

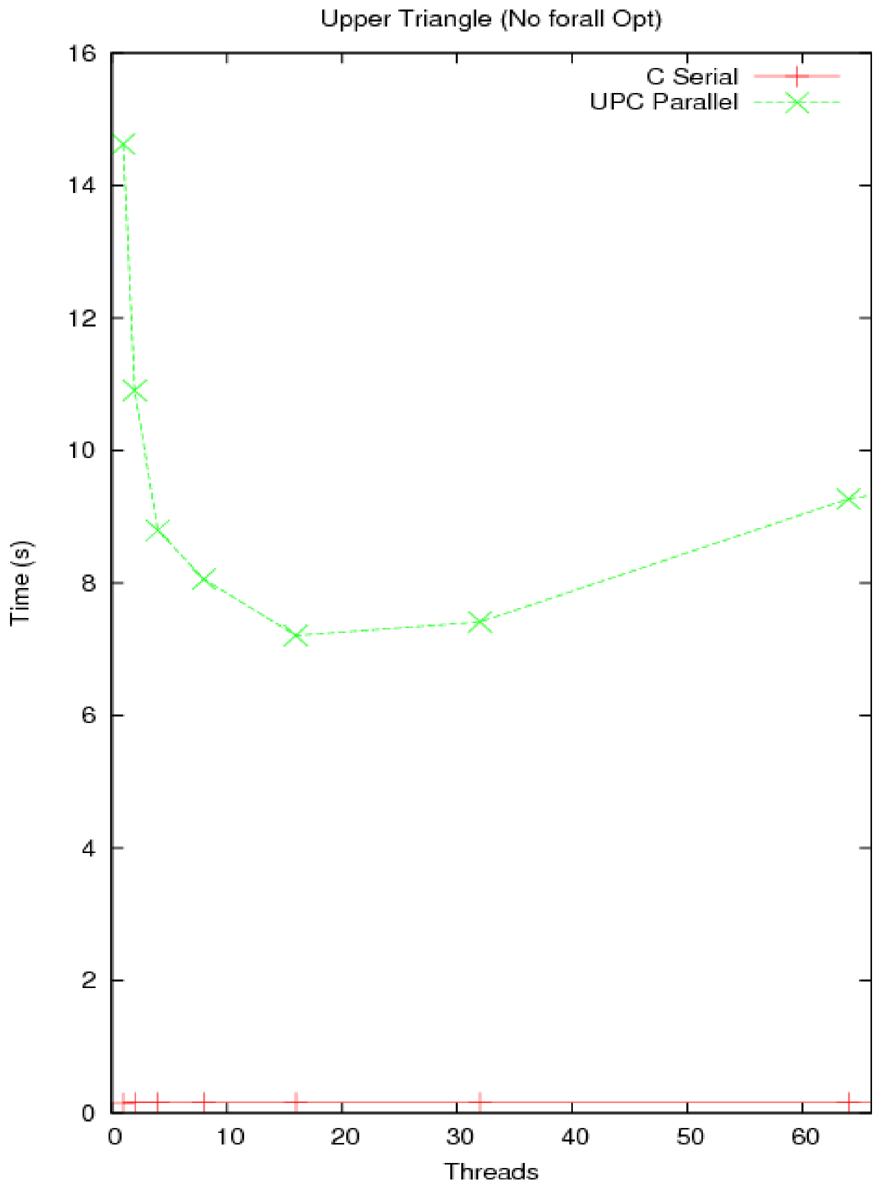
branch

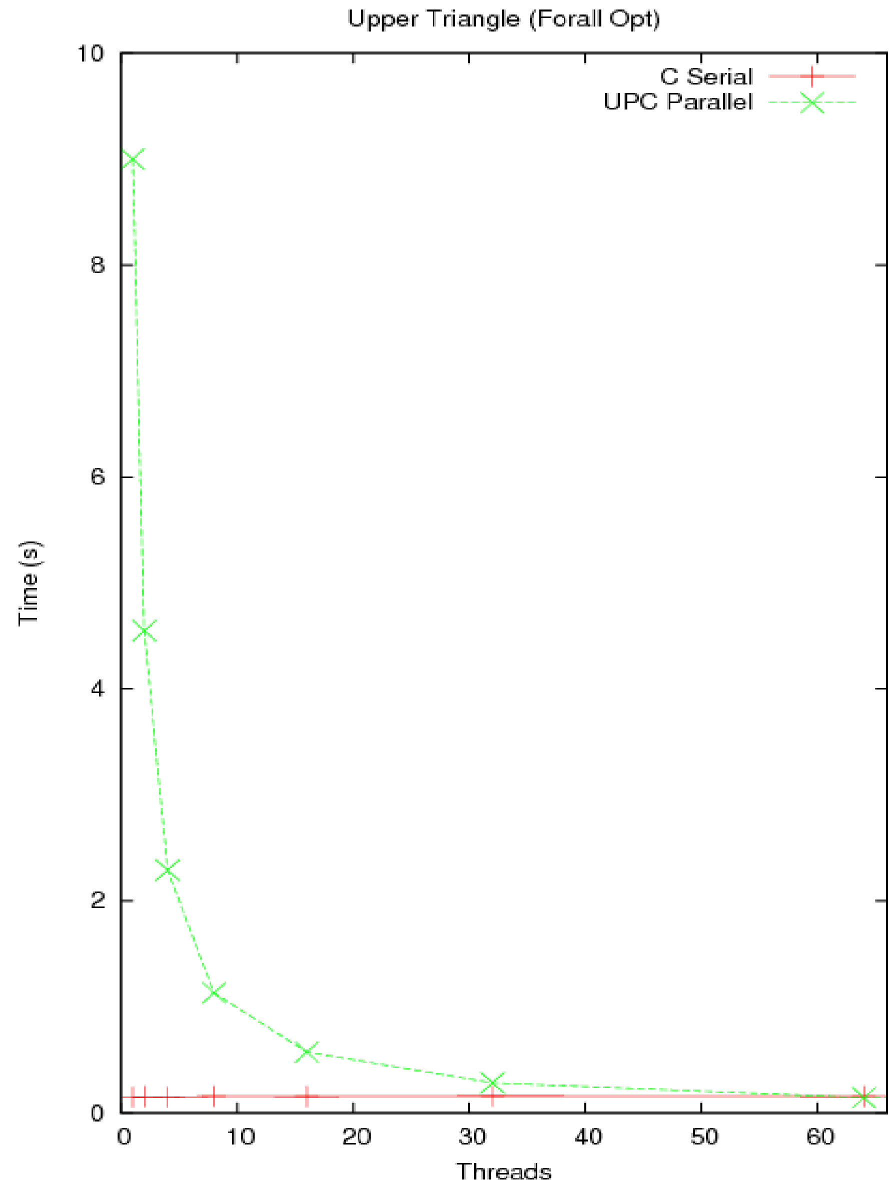
optimized loop

```
shared [BF] int A[N], B[N], C[N];
for (i=MYTHREAD*BF; i<N; i+=THREADS*BF)
    for (j=i; j < i+BF; j++)
        A[j] = B[j] + k*C[j];
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Parallel Loop Reshaping





This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Locality Analysis for shared pointers

```
shared [BF] int B[N], C[N];
int main() {
  upc_forall(int i=0; i<N; ++i; &B[i])
      B[i] = C[i] = i*MYTHREAD;
  upc_barrier;

  shared [BF] int *pA = (shared [BF] int*)
    upc_all_alloc(N/BF*sizeof(int), BF);

  func1(pA, &C[0]);
  func2(pA, &C[0]);
}
void func1(shared int *pA, shared int *pC) {
  upc_forall (i=0; i < N; i++; &B[i])
      pA[i] = B[i]  op  k*pC[i];
}
void func2(shared int *pA, shared int *pC) {
  upc_forall (i=0; i < N; i++; &B[i])
      pA[i+BF] = B[i]  op  k*pC[i+BF];
}
```

Initialize arrays B and C

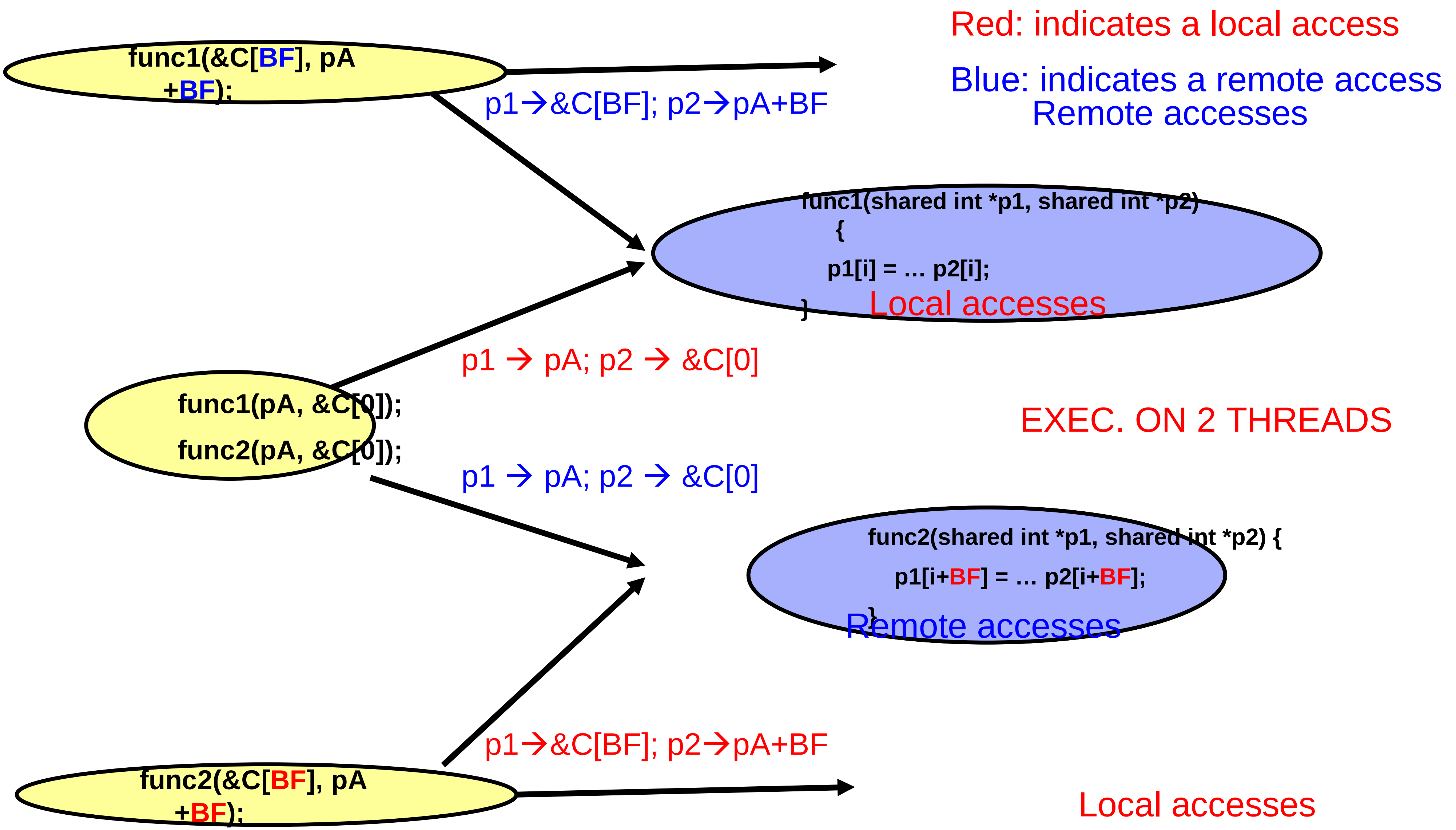
Allocate an array on N integers and blocking factor equal to BF

Pass the address of the 2 arrays

What do we know about the locality of the 2 pointers to shared in the called function ?

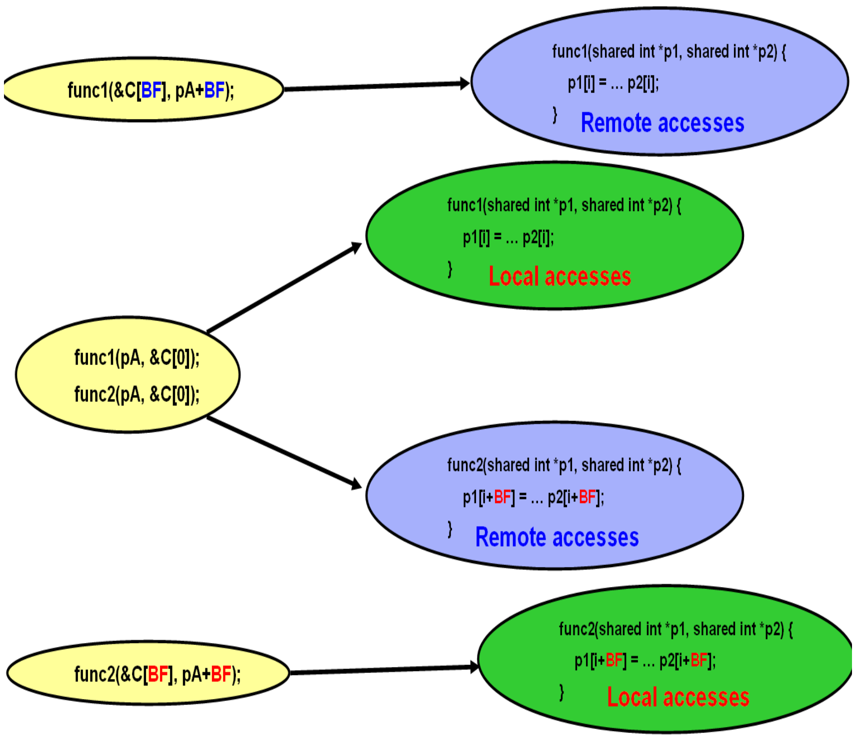
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Possible approach: function cloning



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Possible approach: function cloning



Analysis:

- Create call graph
- Label call edges with arguments
- Merge call edges with same source and target and same actual arguments
- For each edge into a node identify privatization opportunities. If opportunities exist duplicate calee nodes and adjust edges
- Label privatizable shared references in a node

Drawbacks:

- This approach is an all program inter-procedural analysis
- Potentially large code growth: entire function code is cloned

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

A second approach: loop versioning

Objective: privatize pointer-to-shared dereferences

```
shared [BF] int A[N], B[N], C[N];
void foo(shared int *pA, shared int *pC) {
  upc_forall (i=0; i < N; i++; &B[i])
      pA[i] = B[i]  op  k*pC[i];
}
```

- B[i] is a shared local access
- the target of pA, pC is not known until the program executes
- How can we determine the affinity of pA, pB ?

Analysis:

– gather candidate pointer dereferences in the upc_forall loops in the program
– candidate pointers must be loop invariant and have the same blocking factor as the affinity expression
– generate a copy of the original loop (version the loop)
– versioning condition uses UPC runtime calls in order to determine at runtime that pointers point to a shared array with appropriate layout

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Forall Loop Versioning

```
void foo(shared int *pA, shared int *pC) {
   ver_pA = (upc_threadof(pA)==0) ? (upc_phaseof(pA)==0) ? 1 : 0;
   ver_pC = (upc_threadof(pC)==0) ? (upc_phaseof(pC)==0) ? 1 : 0;
   if (ver_pA && ver_pC) {
     upc_forall (i=0; i < N; i++; &B[i])
       pA[i] = B[i]  op  k*pC[i];
   } else if (ver_pA) {
     upc_forall (i=0; i < N; i++; &B[i])
       pA[i] = B[i]  op  k*pC[i];
   } else if (ver_pC) {
     upc_forall (i=0; i < N; i++; &B[i])
       pA[i] = B[i]  op  k*pC[i];
   } else {
     upc_forall (i=0; i < N; i++; &B[i])
       pA[i] = B[i]  op  k*pC[i];
   }
}
```

All 3 references can be privatized

2 references can be privatized

2 references can be privatized

Only reference B[j] can be privatized

Create versioning conditions

Version 1 of the loop: both pointers dereference access local memory

Version 2 of the loop: pA[i] access local memory

Version 3 of the loop: pC[i] access local memory

Original Loop

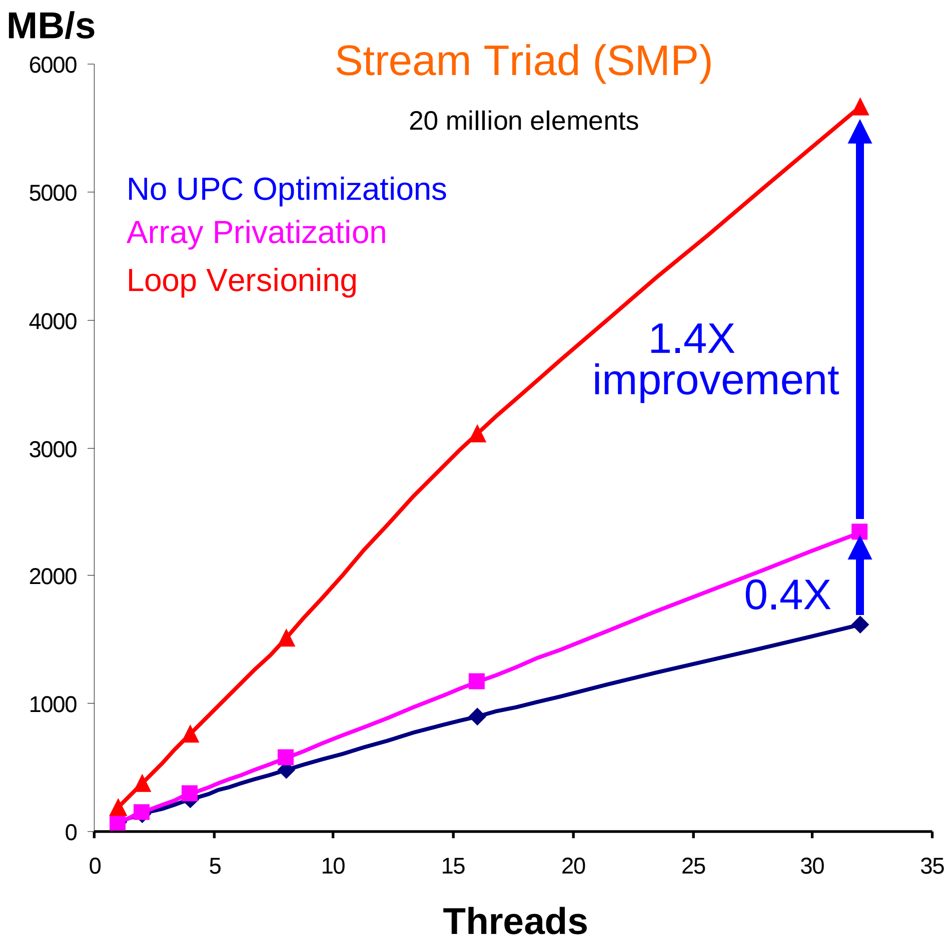
Advantages:

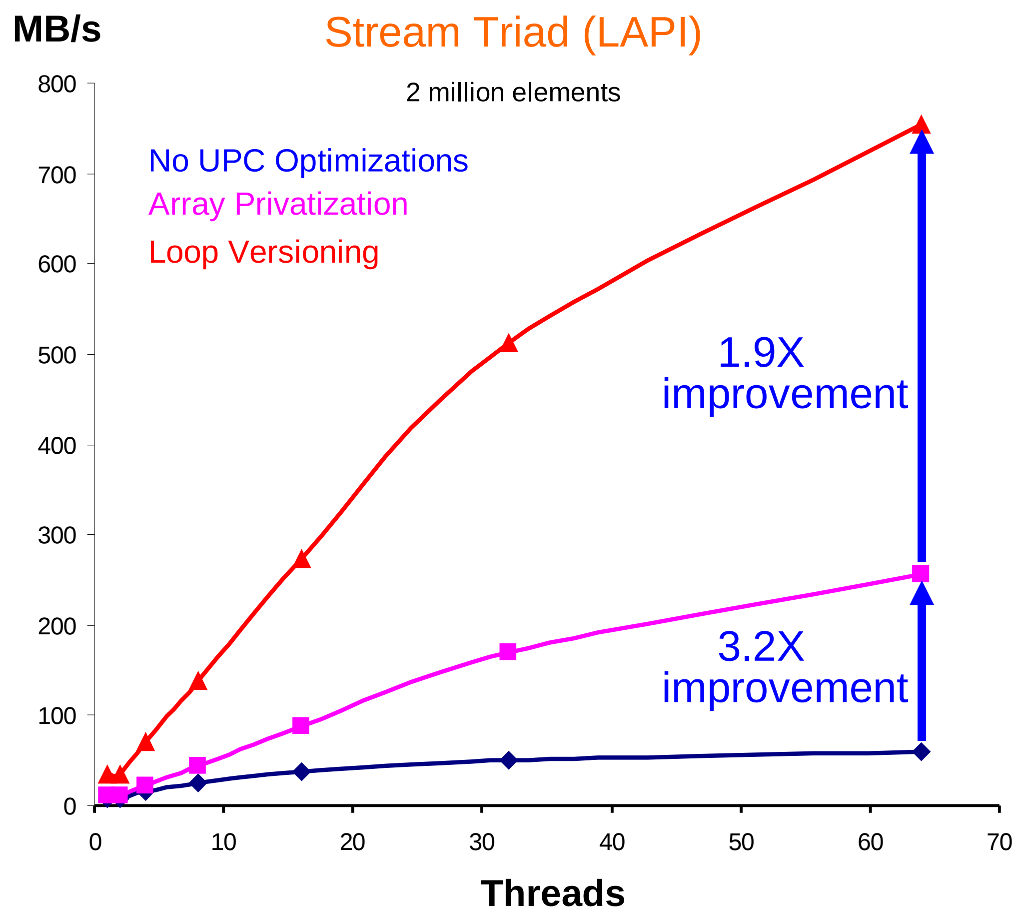
- Intra-procedural loop based analysis. Does not require all program knowledge.
- Code growth limited to upc_forall loops.
- Can limit code growth by using an all or nothing strategy: only one conditional branch is generated.
- Conditional branches can be folded by propagating information from the caller to the callee (inter-procedurally).

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

UPC Forall Loop Versioning

Exploited Locality



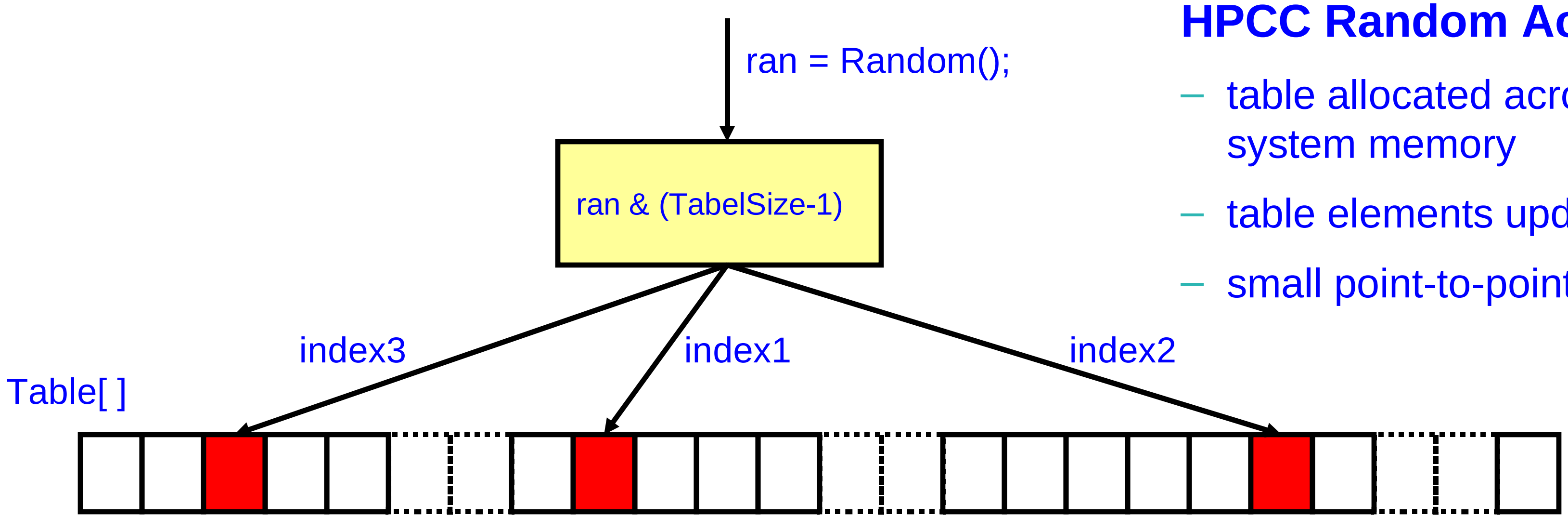


AIX 5.3, Power5, 2.3 GHz

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Remote Update Optimization

```
u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);
upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] ^= ran;
}
```

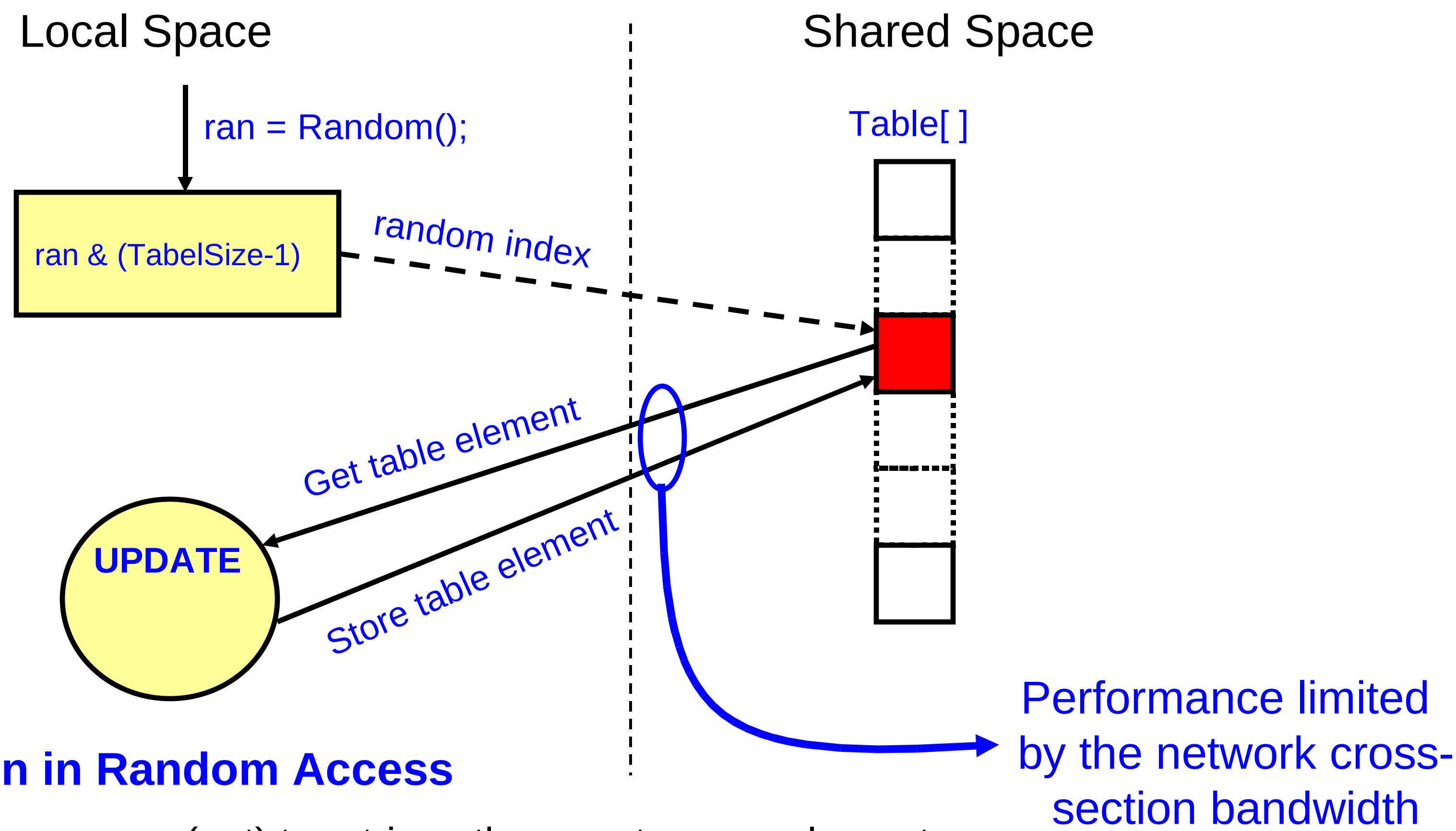


HPCC Random Access

- table allocated across distributed system memory
- table elements updated randomly ☹
- small point-to-point communication ☹

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Remote Update Optimization



Communication in Random Access

– 1 fine grained messages (get) to retrieve the remote array element

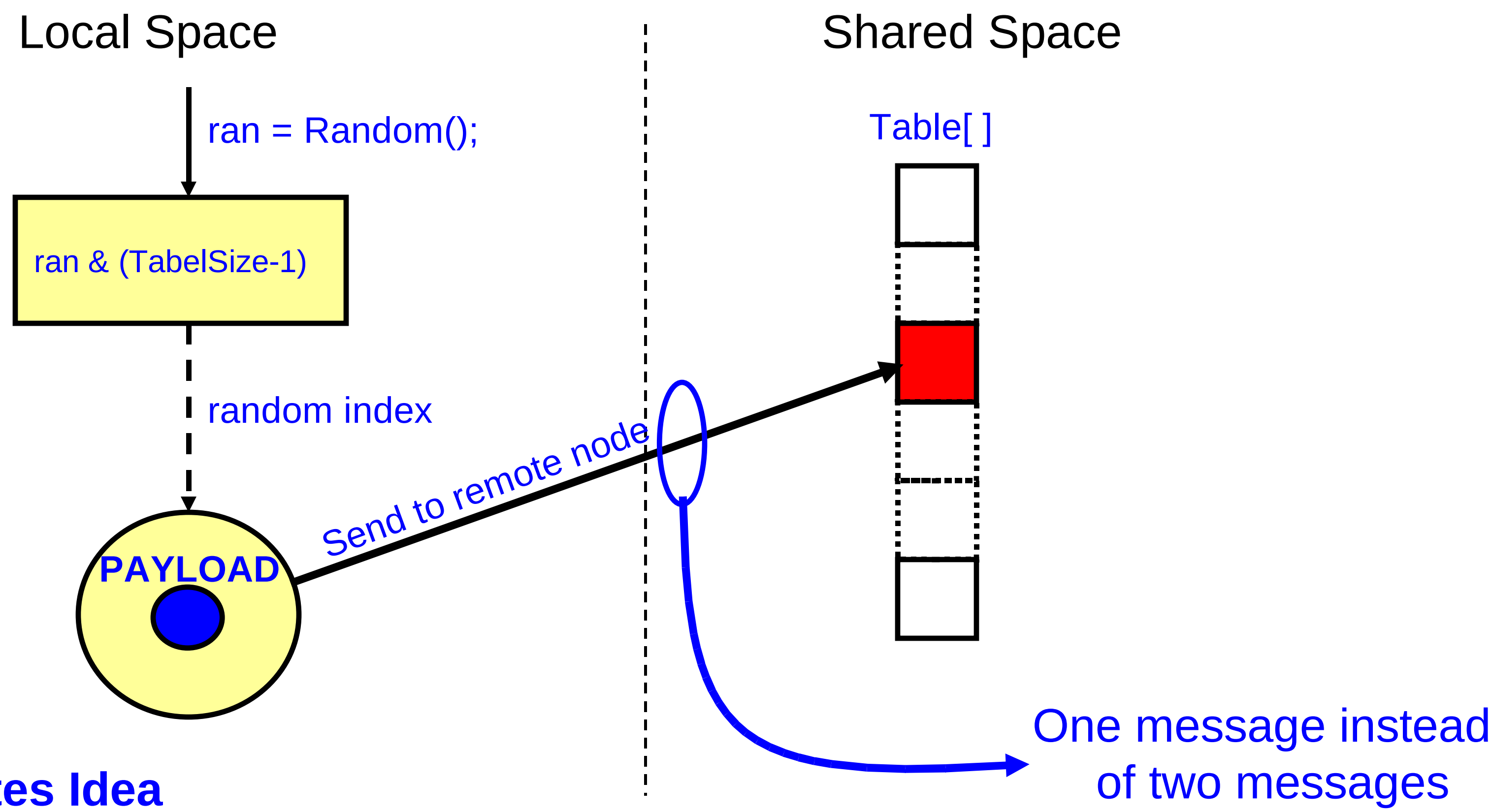
– Update the array element locally

– 1 fine grained message (put) to store the updated value

– Compiler cannot predict whether the access is shared local or shared remote

HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Remote Update Optimization



Owner computes Idea

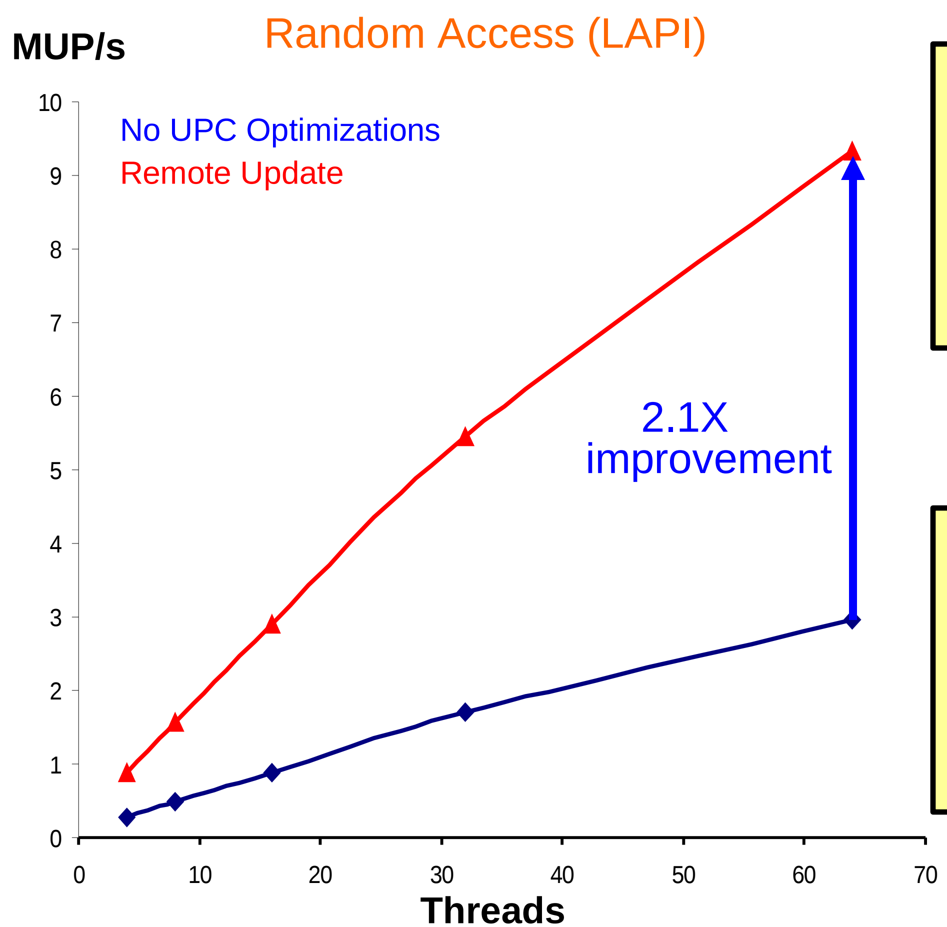
- Compiler recognize update pattern and issues 1 message:

  __xlupc_update(Table_h, index, ran, ^=);

- the update is now done on the **remote** node
- the communication overhead is reduced by half ☺

Remote Update Optimization

Reduced communication




```
u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);
upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] ^= ran;
}
```

```
u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);
upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    __xlupc_update(Table_h, index, ran, ^=);
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Array Idiom Recognition

```
shared [BF] int A[N], B[N], C[N];
int a[N];

int main() {
  if (MYTHREAD == 0) {
    for (int i = 0; i < N; i++)
       A[i] = i;
    upc_barrier;
    for (int i = 0; i < N; i++)
       a[i] = A[i];
    for (int i = 0; i < N; i++)
       B[i] = a[i];
    upc_barrier;
    for (int i = 0; i < N; i++)
       C[i] = B[i];
}
```

Common array initialization idioms:

2. Initialize all shared array elements
3. Copy the shared array to a local array
4. Copy the local array to a second shared array
5. Copies one shared array to another shared array

Observations:

- Each operations requires fine grain communication
- How can we avoid the overhead of many small transfers to/from shared memory ?

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Array Idiom Recognition

```
shared [BF] int A[N];
int a[N];
int main() {
  if (MYTHREAD == 0)
    for (int i = 0; i < N; i++)
      A[i] = a[i];
}
```

```
shared [BF] int A[N];
int a[N];
int main() {
  if (MYTHREAD == 0)
    for (i = 0; i < N; i+=BF)
      upc_memput(&A[i], &a[i], BF*sizeof(a[i]));
}
```

Analysis:

2. Collect stride one accesses to shared array
3. Only non-strict accesses are considered
4. Classify the pattern into one of:
   - upc_memset: write same value to shared array
   - upc_memget: read from shared array to local array
   - upc_memput: write to shared array from local array
   - upc_memcpy: write from shared array to shared array
5. Generate the upc_mem* call and adjust the loop increment.

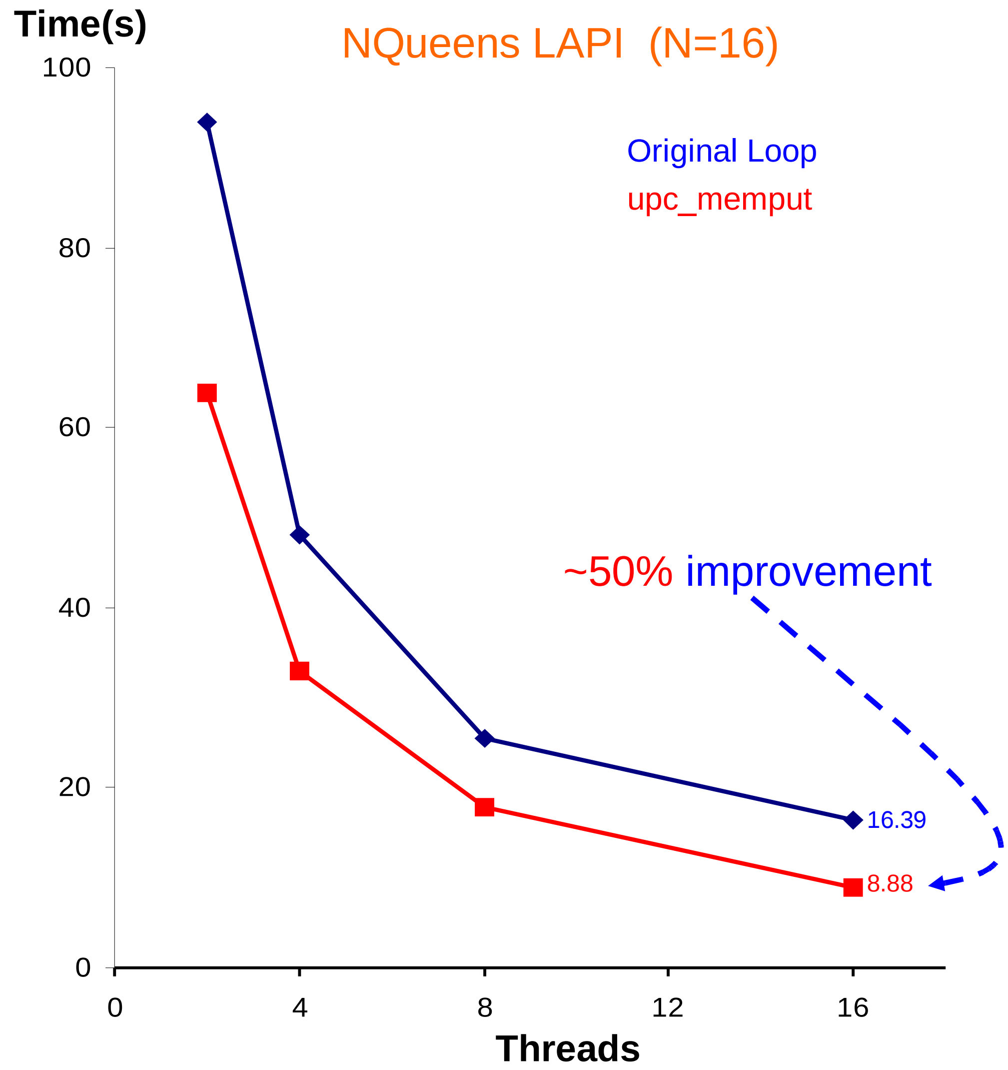
Extensions:

```
A[i+k] = a[i]  // how do we handle the offset ?

for(i=0; i<N; ++i) {
  if (i%BF < k)
    A[i] = a[i];  // first k accesses in block
  if (i%BF == k)
    upc_memput(&A[i], &a[i], (BF-k)*sizeof(a[i]));
}
```

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Array Idiom Recognition

Reduced communication



AIX 5.3, Power5, 2.3 GHz

```
shared [MAXTS] long int * finalPtr =
&FINALSOL[offset[MYTHREAD]];
step = SOLUT[MYTHREAD];
for (p = 0; p < step; p++)
  finalPtr[p] = ptr[p];
```

```
shared [MAXTS] long int * finalPtr =
&FINALSOL[offset[MYTHREAD]];
step = SOLUT[MYTHREAD];
for (p = 0; p < step; p+=MAXTS)
    upc_memput(finalPtr, ptr, …);
```

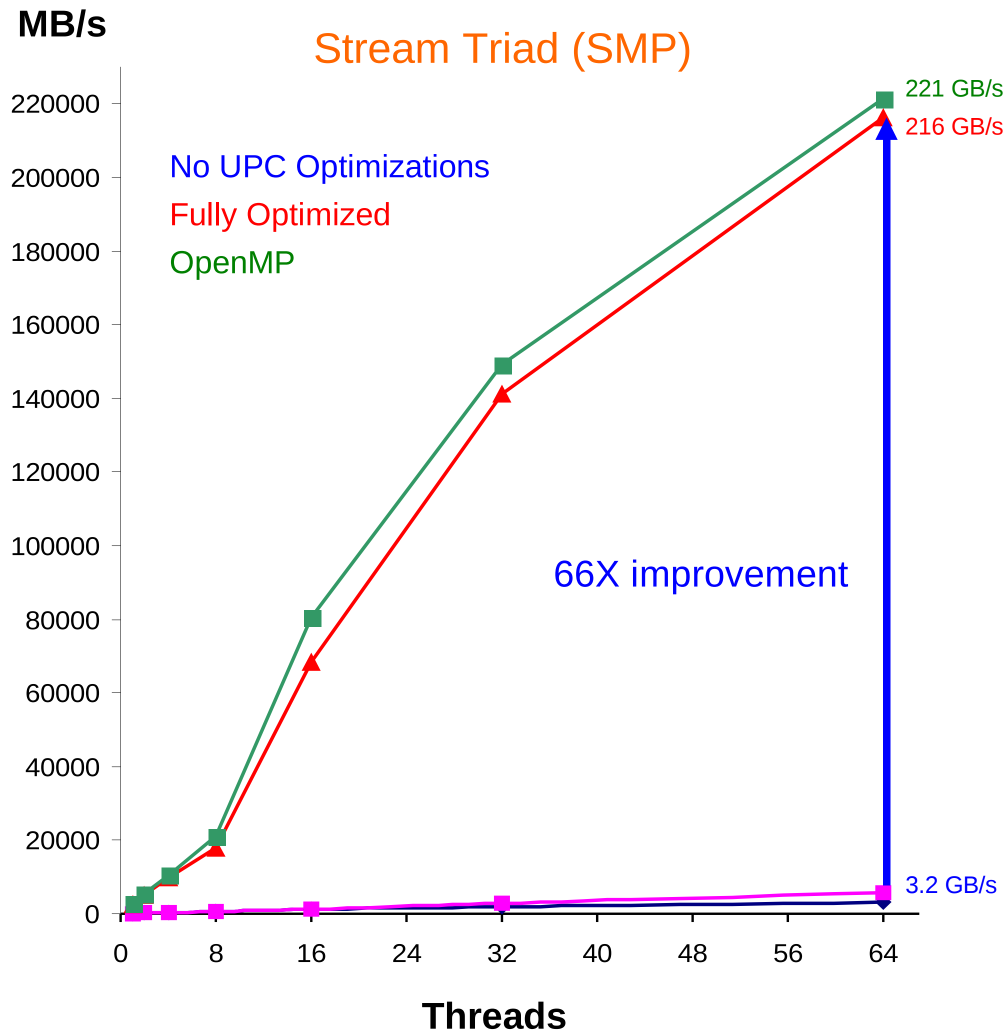
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

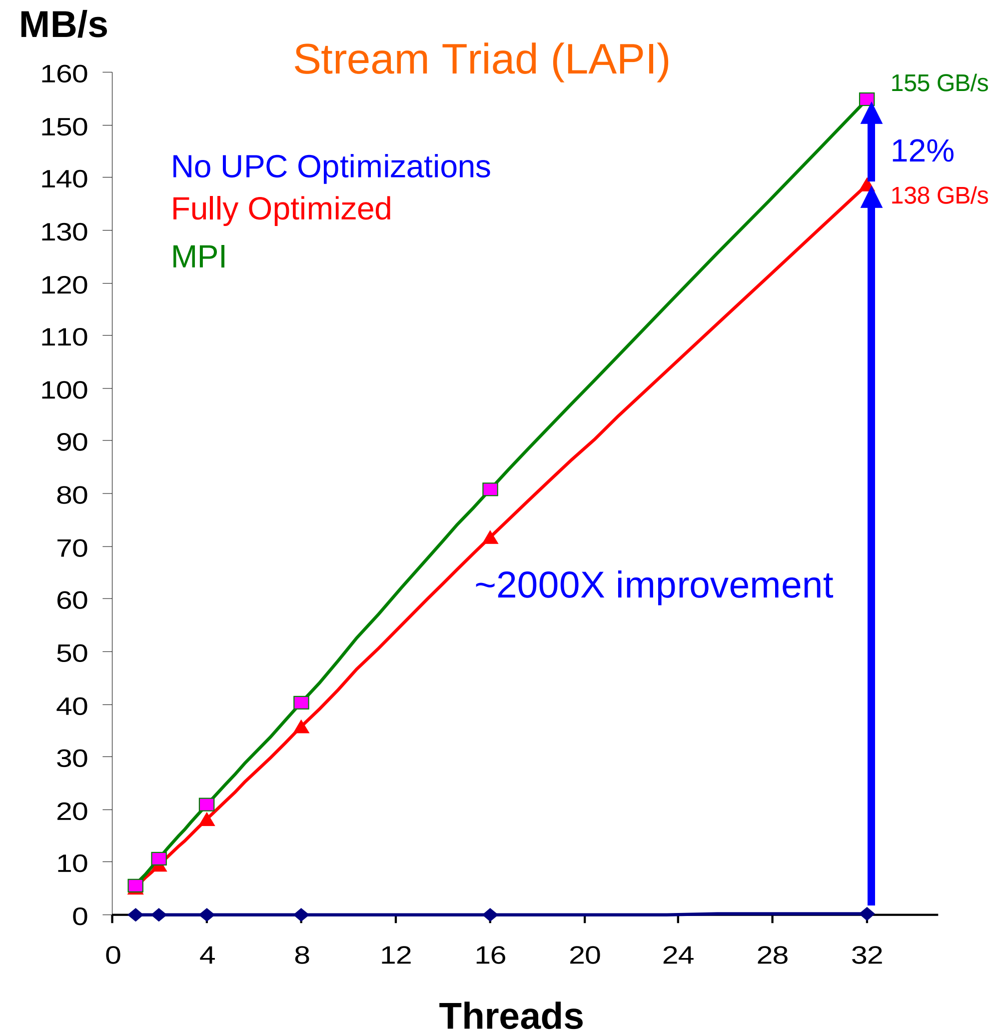
Locality Optimizations and Forall Optimizations

- **Each of the optimizations presented has a reasonable impact on performance**
  - Reduce overhead of accessing shared data
  - Reduce overhead of executing parallel loop
- **However, when all optimizations are combined, they have a dramatic impact on performance**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

HPCC Stream Triad

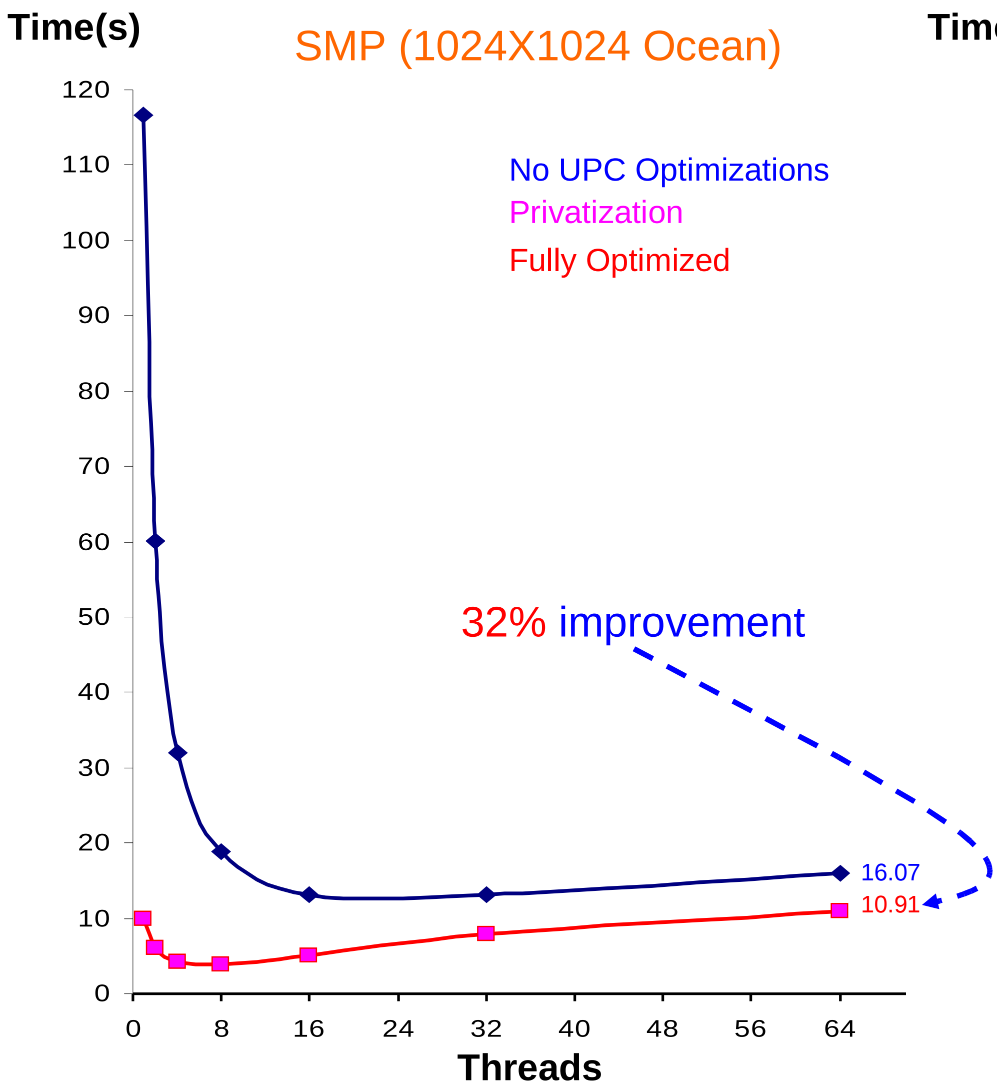


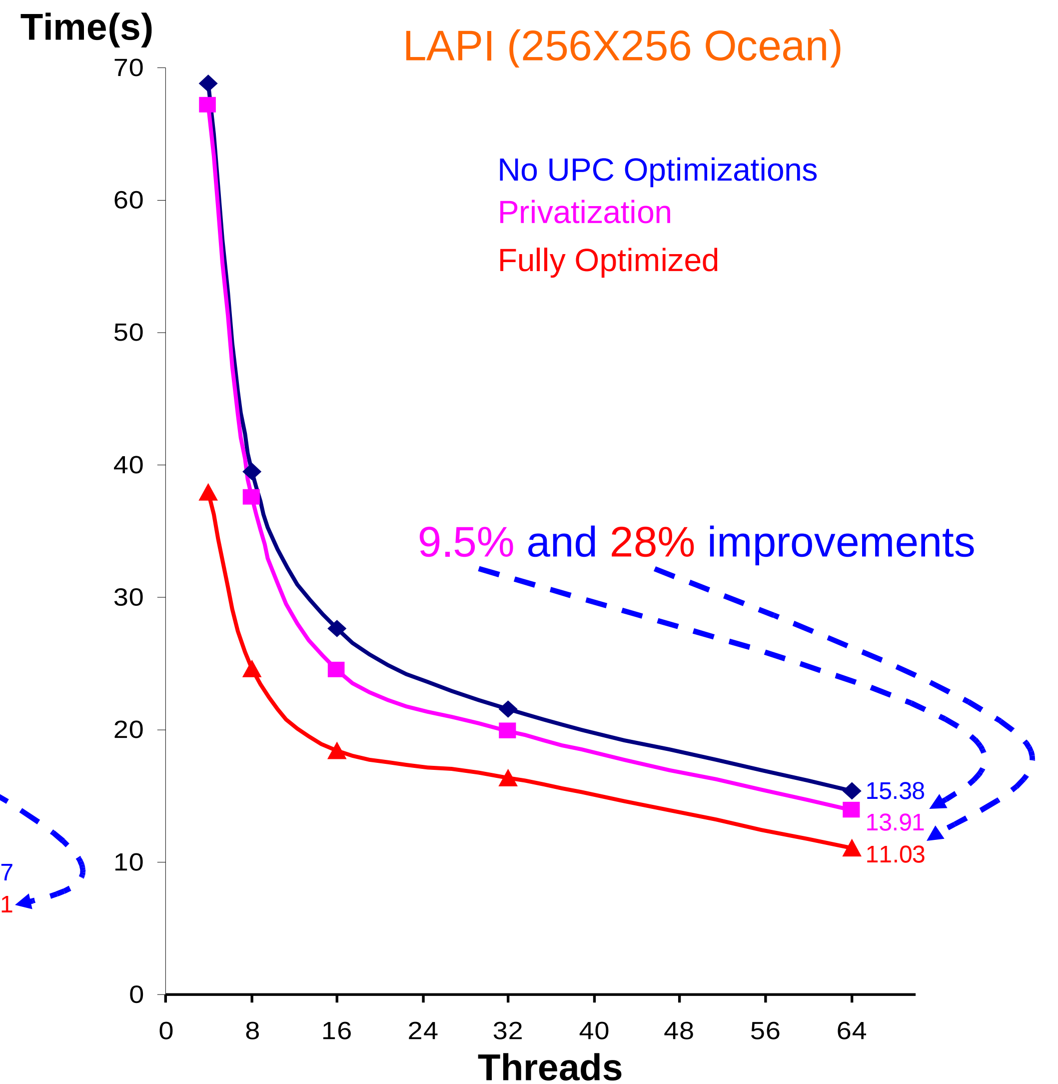


AIX 5.3, Power5, 2.3 GHz

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

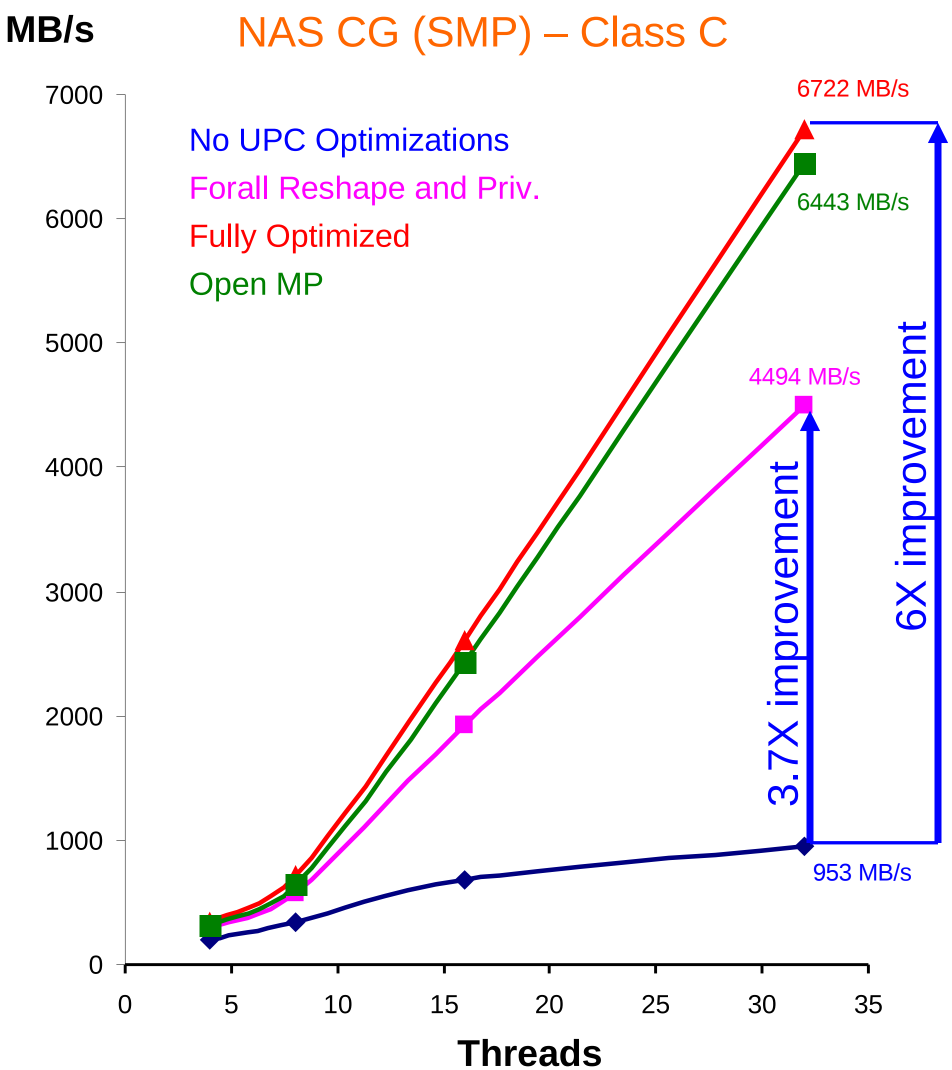
UPC Fish - Predator-prey model

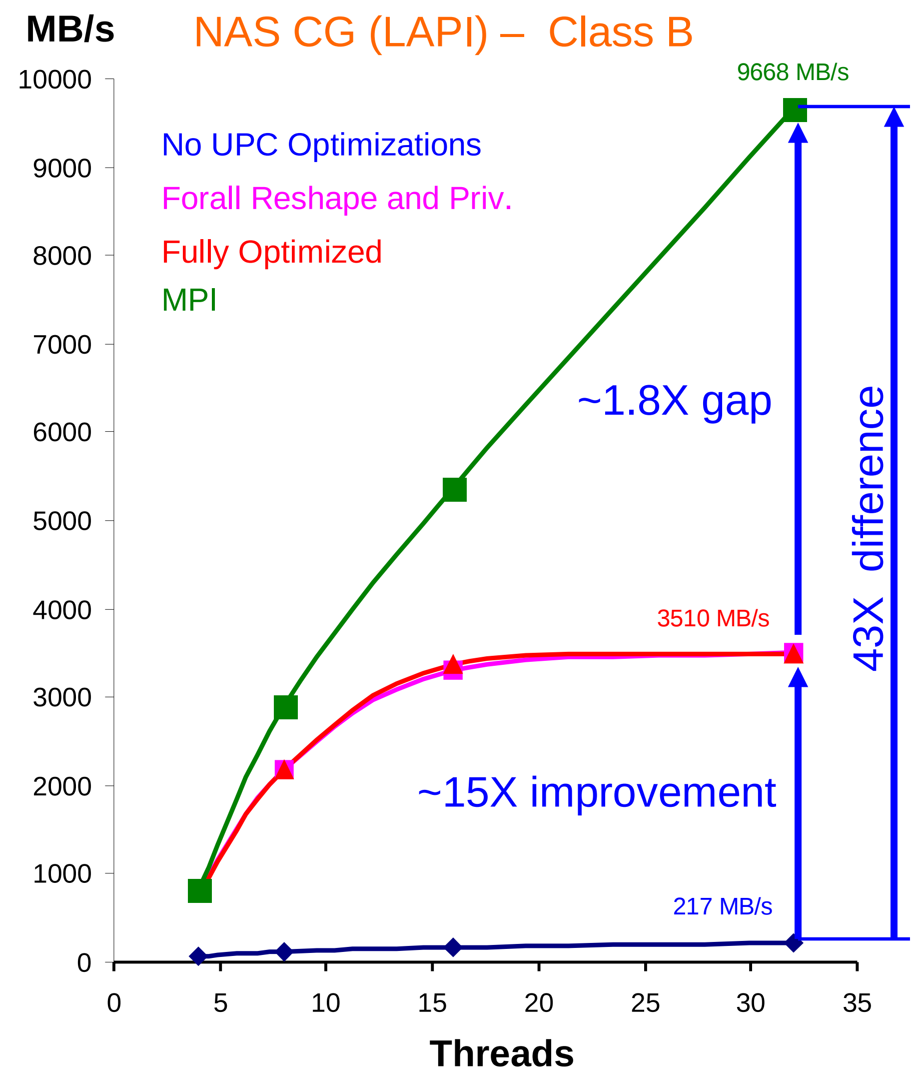




AIX 5.3, Power5, 2.3 GHz

NAS 3.2 CG – UPC vs OMP and MPI





AIX 5.3, Power5, 2.3 GHz

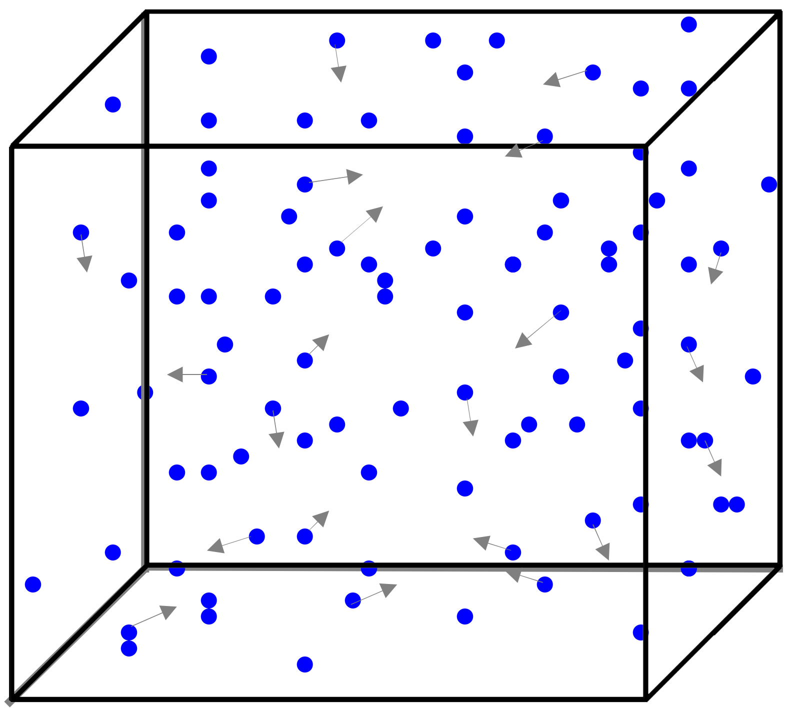
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

4. Examples of performance tuning

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Molecular Dynamics

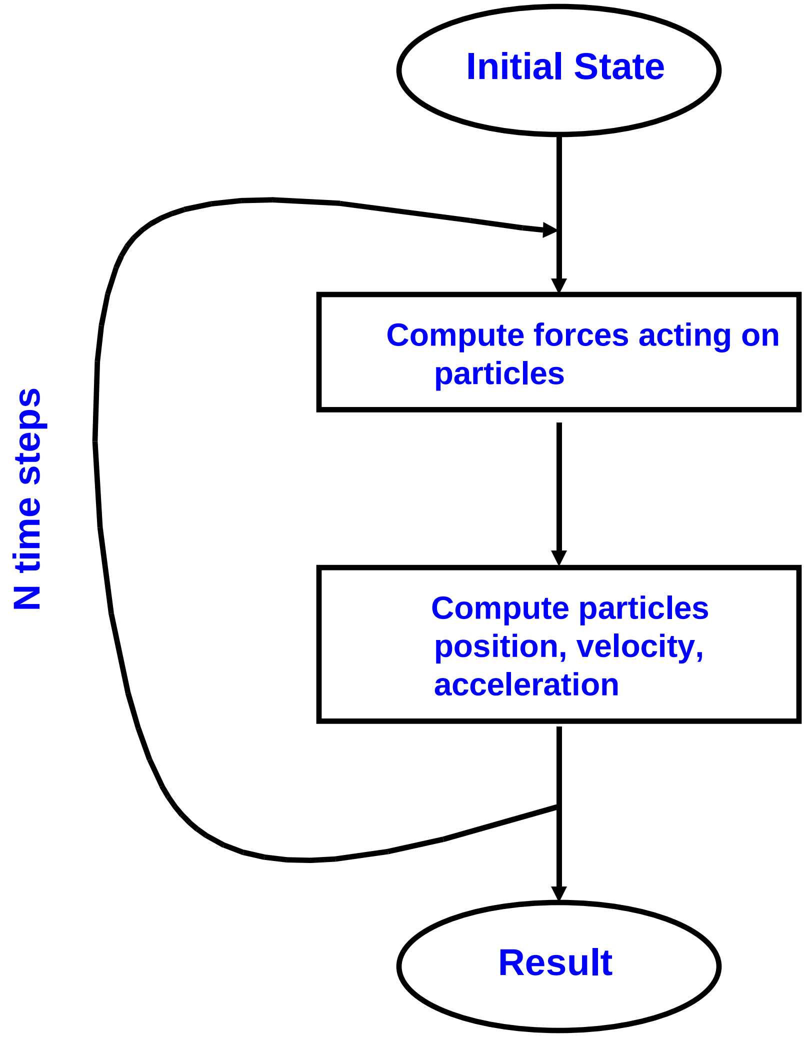
- N particles in 3D space interact with each other
- Compute particles new position, velocity, acceleration at each time step



- GOOD:
  - **Given force acting on each particle the computation of particles position, velocity, acceleration is an embarrassing parallel problem**
- BAD:
  - **Force acting on particle p.f[i] is a function of the gravitational attraction of all other particles …**

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Molecular Dynamics



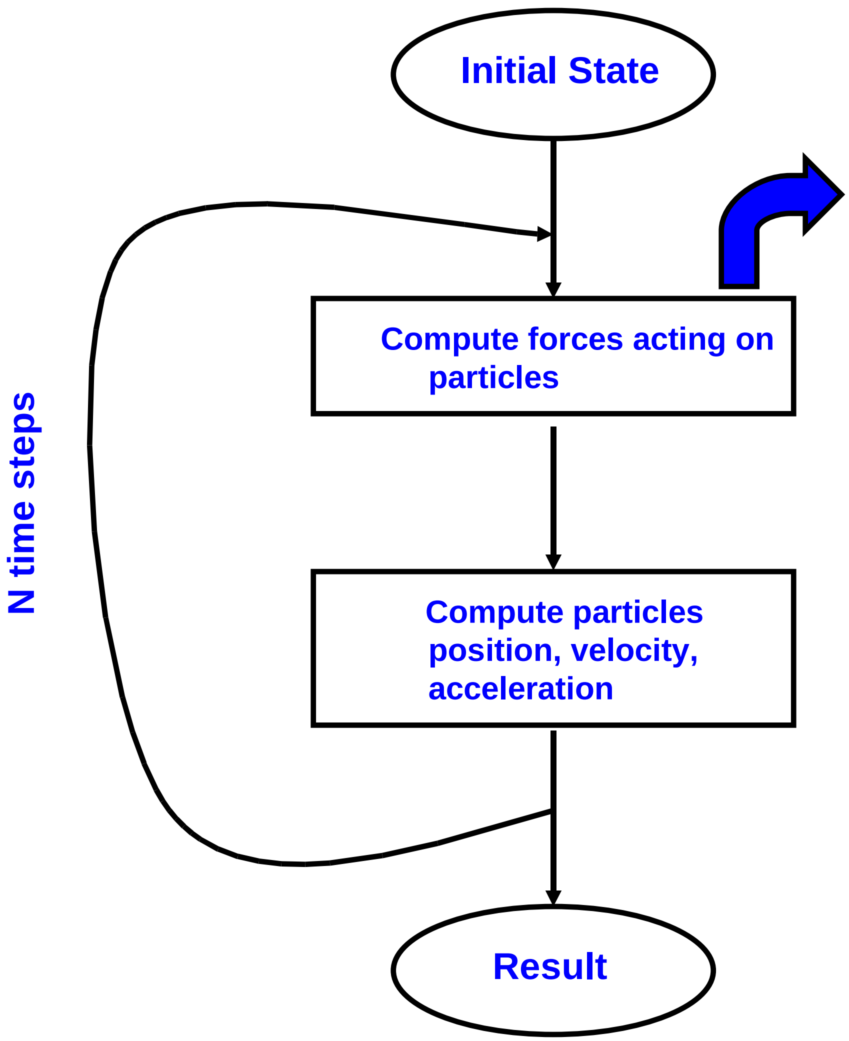

```
typedef struct {
  double p[3];   // particle position
  double v[3];   // particle velocity
  double a[3];   // particle acceleration
  double f[3]; // force acting on particle
} particle_t;

#define BF (NPARTS/THREADS)

shared [BF] particle_t PARTS[NPARTS];
shared [BF] double POT[NPARTS];
shared [BF] double KIN[NPARTS];
```

All Local access ☺

```
upc_forall(int i = 0; i < NPARTS; i++; &PARTS[i])
  for(int j = 0; j < NDIM; j++) {
    PARTS[i].p[j] += PARTS[i].v[j]*dt + 0.5*dt*dt*PARTS[i].a[j];
    PARTS[i].v[j] += 0.5*dt*(PARTS[i].f[j]*rmass + PARTS[i].a[j]);
    PARTS[i].a[j] = PARTS[i].f[j]*rmass;
  }
```

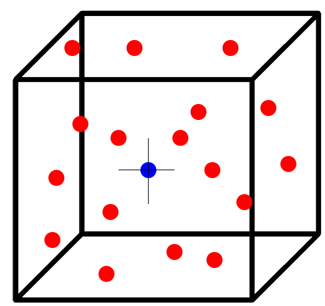
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Molecular Dynamics



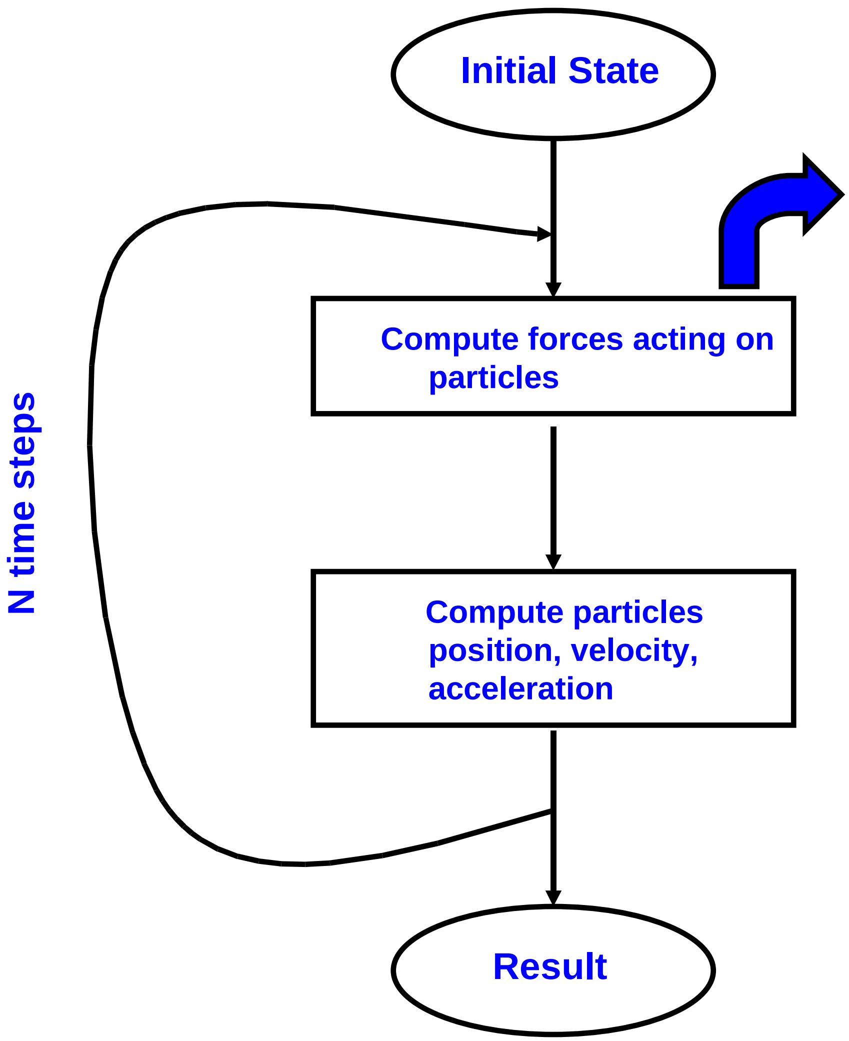

```
upc_forall (i = 0; i < NPARTS; i++; &PARTS[i]) {
  …
  for (j = 0; j < NPARTS; j++) {
    if (i != j) {
      d = dist(&PARTS[i].p[0], &PARTS[j].p[0], rij);
      POT[i] += 0.5*V(d);
      for (k = 0; k < NDIM; k++)
        PARTS[i].f[k] = PARTS[i].f[k] - rij[k]*DV(d)/d;
    }
  }

double dist (shared double *r1, shared double *r2, double *dr) {
  for (int i=0; i < NDIM; i++) {
    dr[i] = r1[i] - r2[i];
    d += dr[i]*dr[i];
  }
  return sqrt(d);
}
```

Remote accesses



This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

Molecular Dynamics




```
upc_forall (i = 0; i < NPARTS; i++; &PARTS[i]) {
  particle_t *p = (particle_t*) &PARTS[i];
  ...
  for (j = 0; j < NPARTS; j++) {
    if (i != j) {
      d = dist_local(&p->pos[0], &PARTS[j].pos[0], rij);
      POT[i] = POT[i] + 0.5*V(d);
      for (k = 0; k < NDIM; k++)
        p->f[k] = p->f[k] - rij[k]*DV(d)/d;
    }
  }
```

Local access

```
double dist _local (double *r1, shared double *r2, double *dr) {
  for (int i=0; i < NDIM; i++) {
    dr[i] = r1[i] - r2[i];
    d += dr[i]*dr[i];
  }
  return sqrt(d);
}
```

Further Improvements:

– Prefetch r2[i] with upc_memget ?

– Prefetch PARTS[j] in caller ?

HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.

6. Conclusions

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Conclusions

UPC = Performance + Productivity

Performance

- Exploitation of data locality
- Coalescing of communication
- Overlapping communication and computation
- One-sided communication
- Optimized collective library

Productivity

- Simple syntax based on C
- Easy partitioning of shared data
- Work-sharing construct with locality information
- No explicit need to manage communication with function calls
- Simple thread synchronization

http://www.alphaworks.ibm.com/tech/upccompiler

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect
the views of the Defense Advanced Research Projects Agency.