



IBM Toronto Laboratory

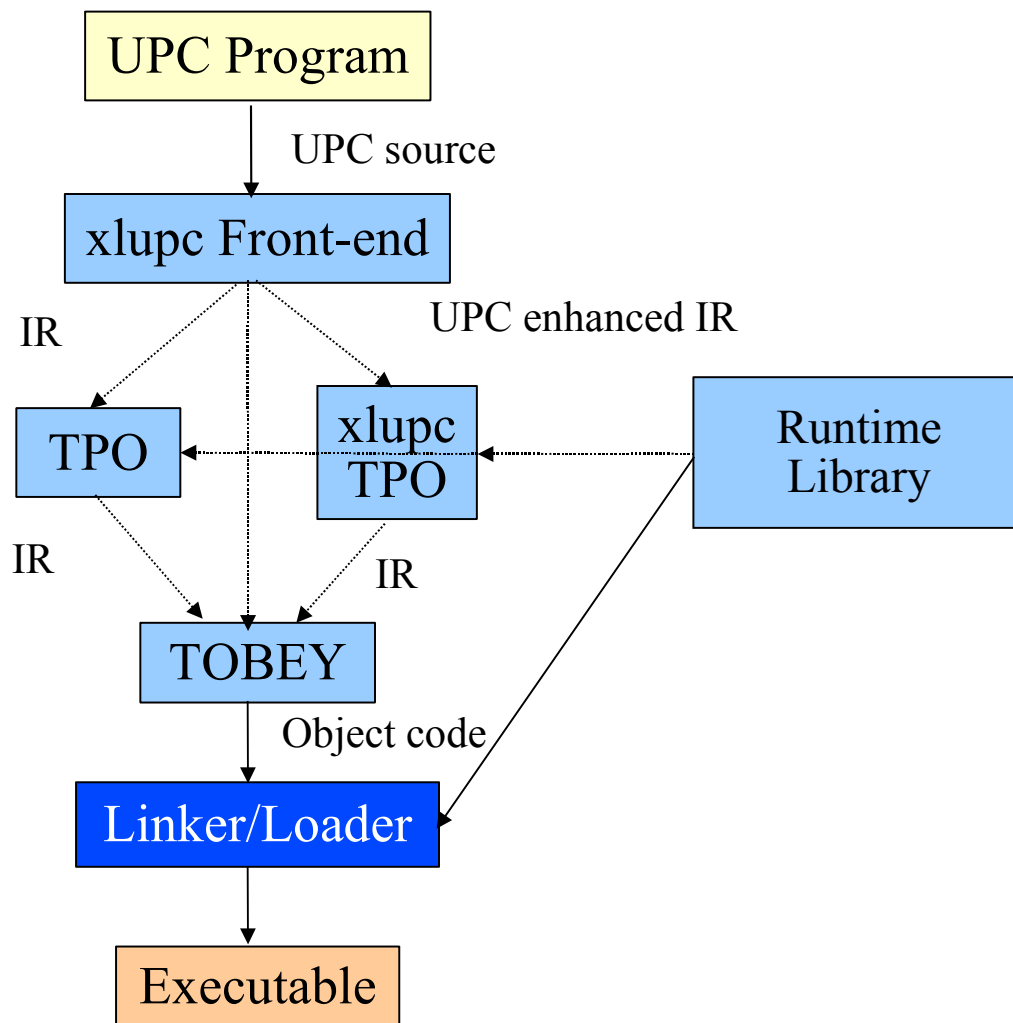
Challenges to Compiling and Optimizing for Unified Parallel C

Ettore Tiotto – IBM Toronto Laboratory
Christopher Barton – University of Alberta

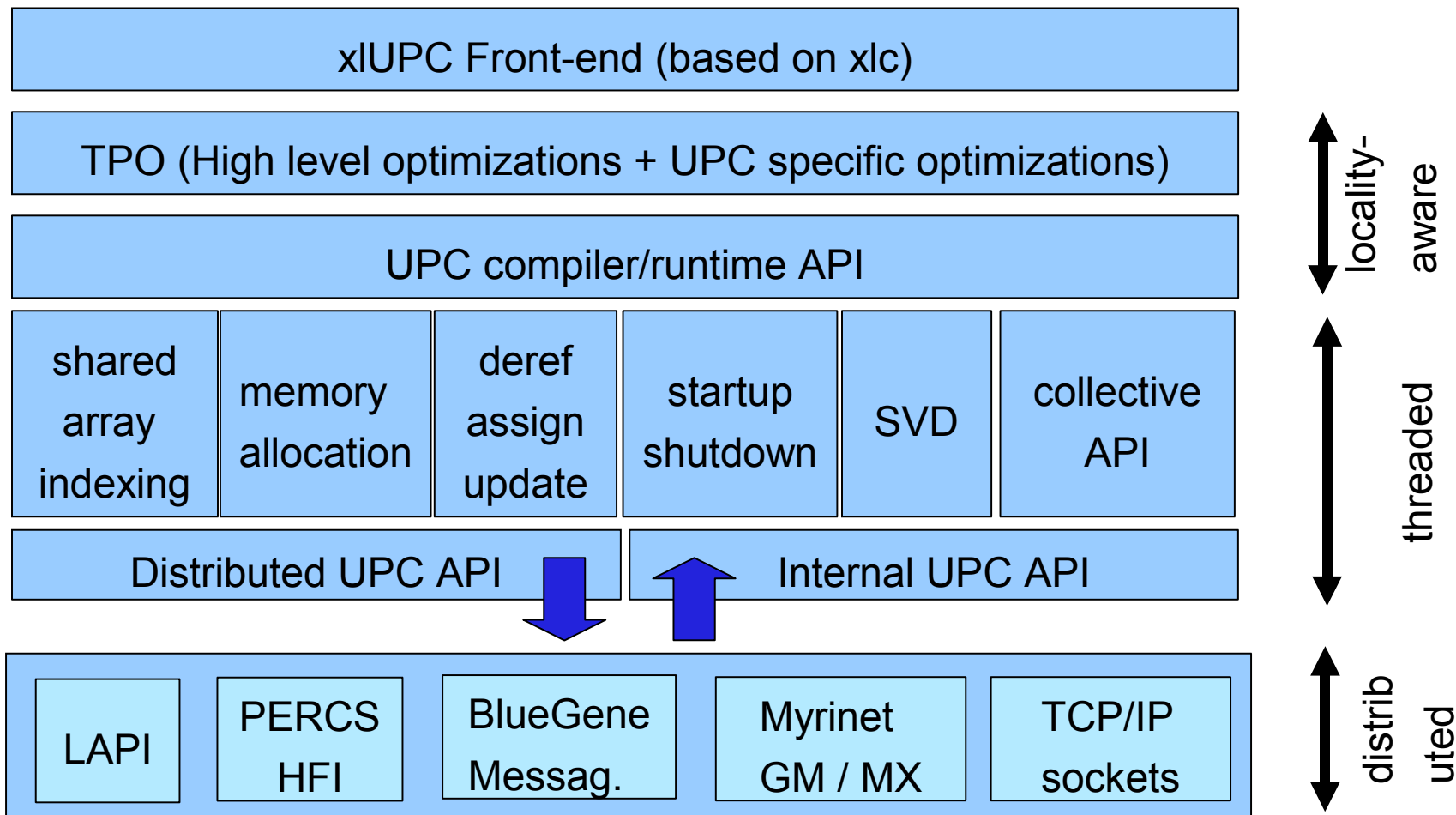
Outline

- **Overview of XLUPC architecture**
- **UPC shared array distribution**
- **Important optimizations for PGAS languages**
 - Parallel loop reshaping
 - Privatization of local shared accesses
 - Reducing communication cost
 - Overlapping communication and computation
- **Initial NAS UPC results**
- **Q&A**

xlupc Compiler Architecture



UPC hybrid runtime stack

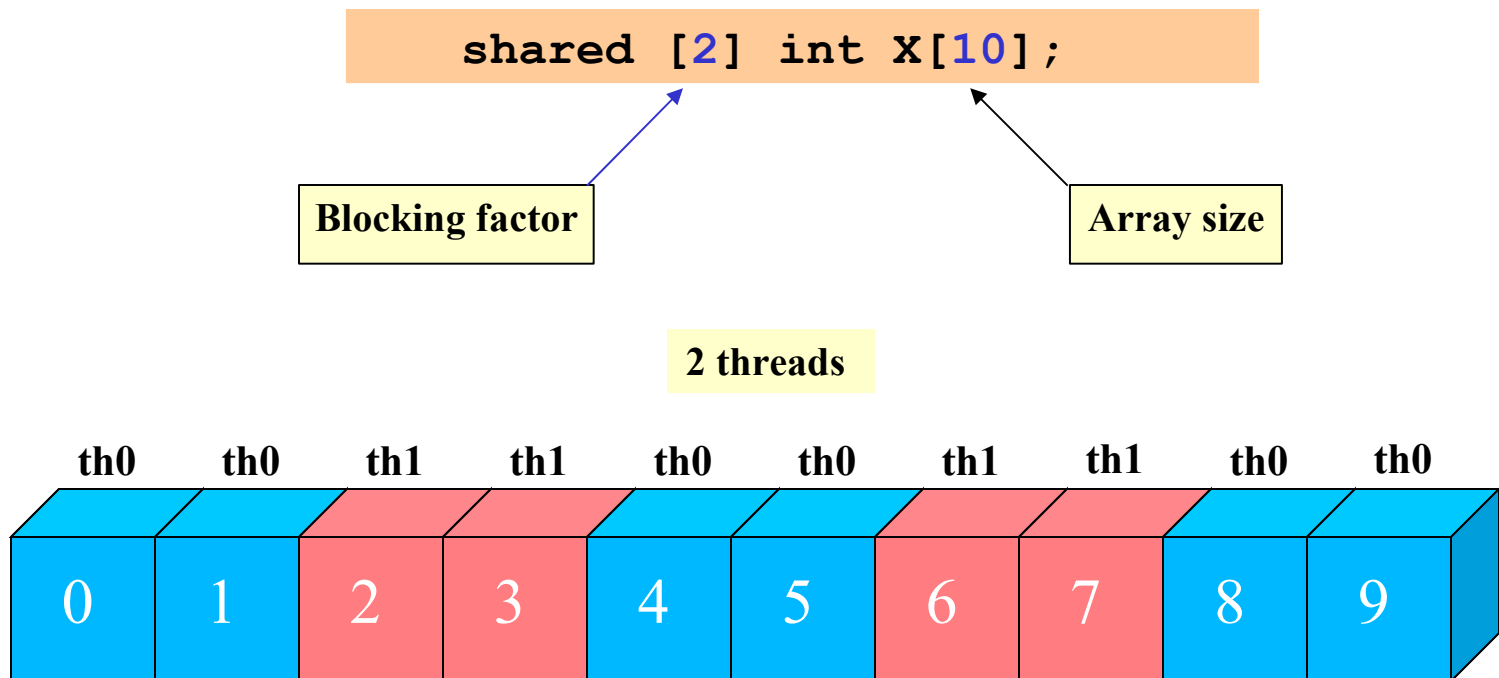


Unified Parallel C (UPC)

- **Partition Global address Space**
 - Allow the declaration of shared variables
 - Programmer specifies shared data distribution
 - Work-sharing construct used to distribute loop iterations
 - Synchronization primitives: barriers, fences, and locks
- Data can be private, shared local and shared remote
 - shared data can be accessed by all threads
 - shared space is divided in portions with affinity to a thread
 - Latency to access *local* shared data is lower than latency to access *remote* shared data
- All threads execute the same program (SPMD style)

Distribution of a shared array in UPC

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

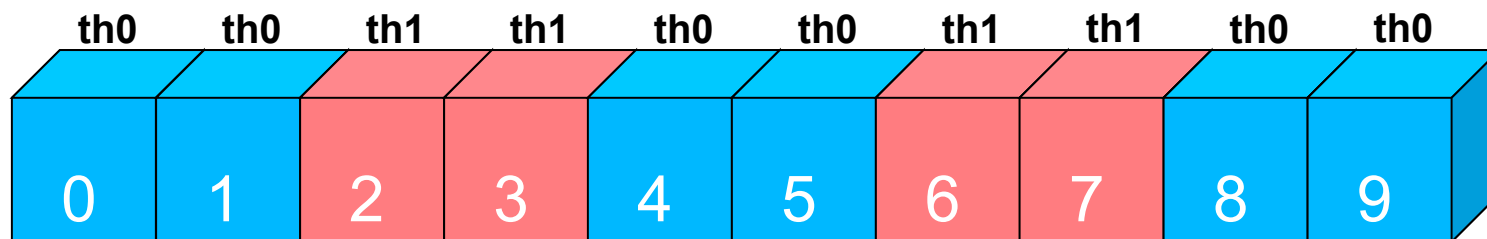


Physical layout of shared arrays

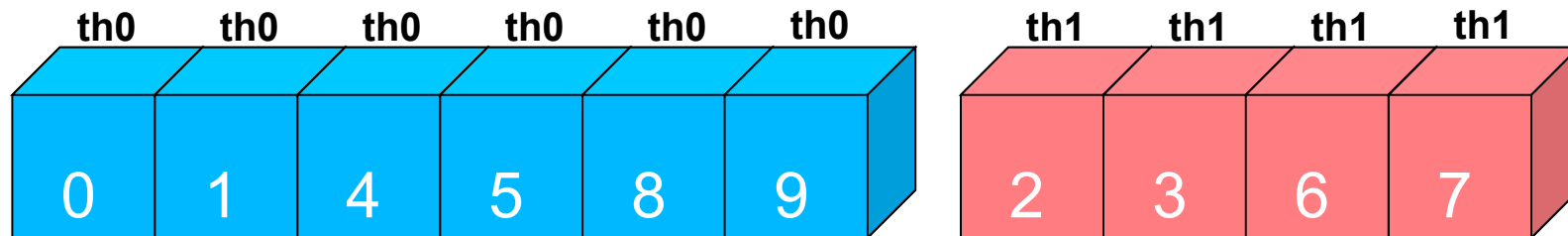
```
shared [2] int X[10];
```

2 threads

Logical Distribution

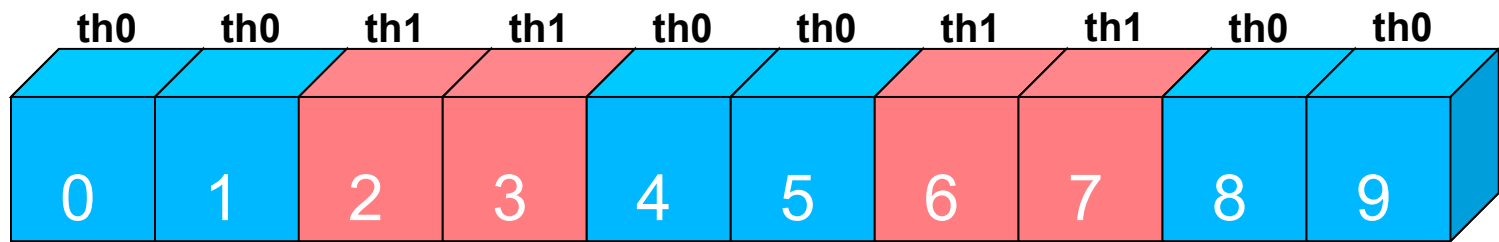


Physical Distribution



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

Examples

`upc_threadof(&a[2]) = 1`

`upc_threadof(&a[5]) = 0`

`upc_phaseof(&a[2]) = 0`

`upc_phaseof(&a[5]) = 1`

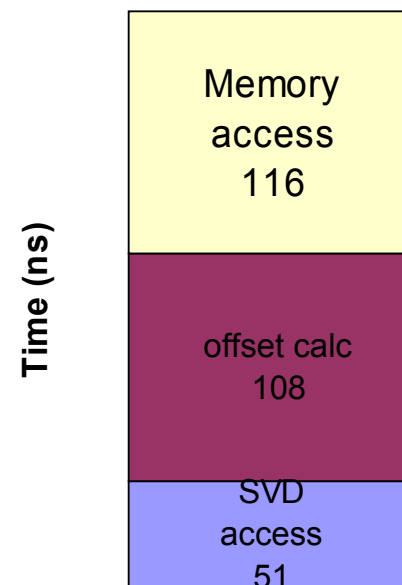
Anatomy of a shared access

```
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



The upc_forall work-sharing construct

- Similar to C for loop, 4th field indicates the affinity
- Thread that “owns” elem. A[i] executes iteration
- Affinity test exec. on *each* iteration by all threads

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

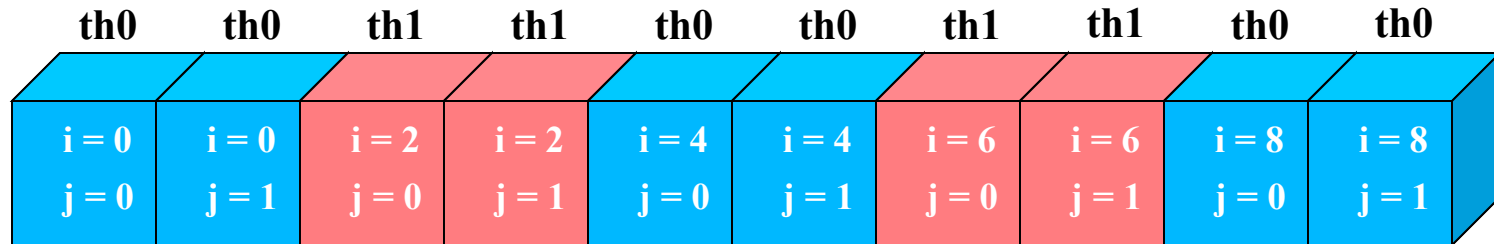
affinity test

```
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];  
}
```

branch

Parallel loop reshaping

- Iteration space is partitioned
- loop nest executed by thread with affinity to &A[i]
- Inner loop iterates through each block element



```
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[i]+C[i];
```

```
}
```

Block by 2

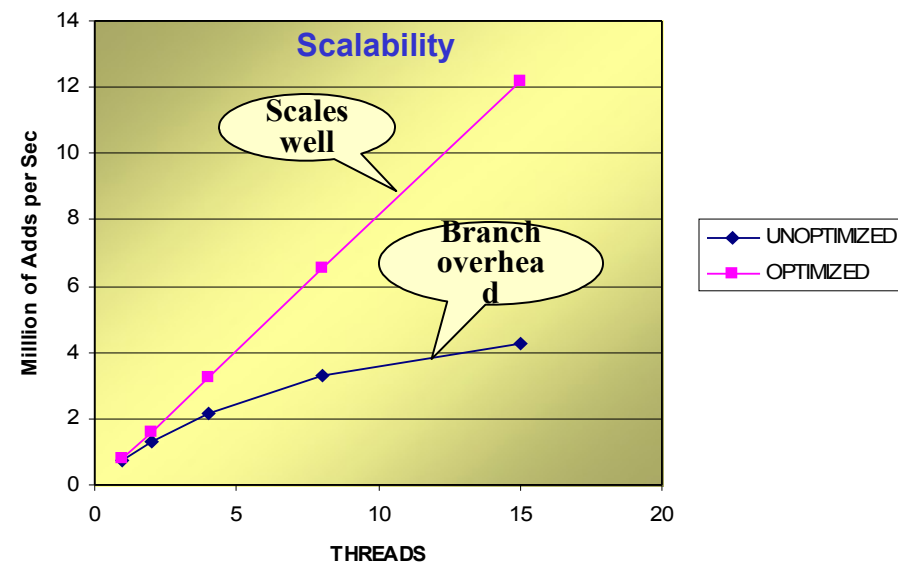
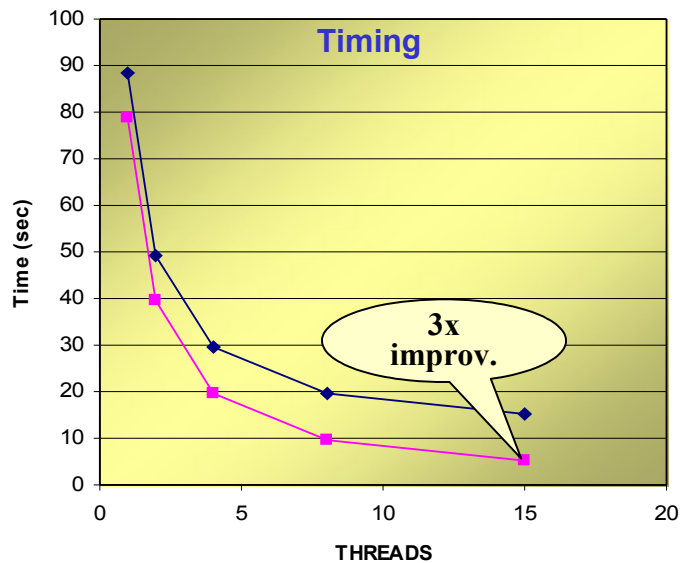
2 threads

Parallel loop reshaping results

- vector add: 64000 elems, block by 32, 1000 iter.

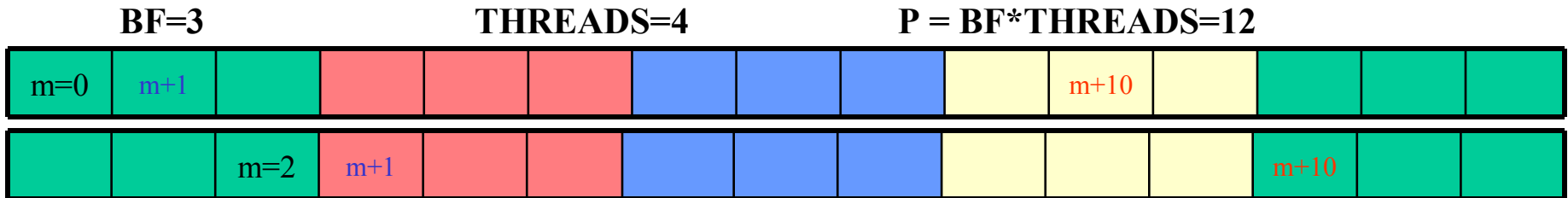
```
upc_forall (i=0; i<N; i++; &RES[i])
  RES[i] = V1[i] + V2[i];
```

```
for (i=B*MYTHREAD; i<N; i+=B*THREADS)
  for (j=i; j<i+B; j++) RES1[j] = V1[j] + V2[j];
```



64 Million adds on PWR4 SMP

Affinity Analysis



- Compute distance vector from affinity expr. for each shared reference
- $A[m+k]$ is local when $(m\%BF) < BF-(k\%BF)$
- Compiler splits accesses into local/remote
- Local accesses are privatized

```
shared [ 3 ] int A[16] ;
```

```
upc forall (m=0; m < Z ; m++; &A[m] )
{
    A[m+1] = m*2;
    A[m+10] = m-3;
}
```



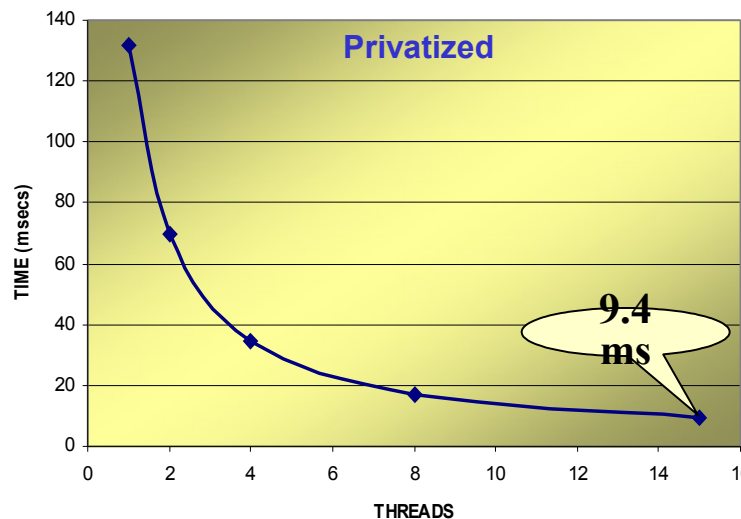
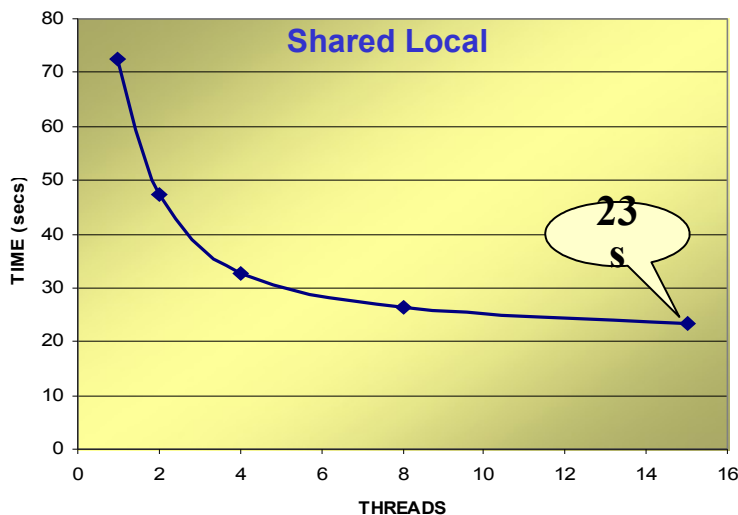
```
upc forall (m=0; m < Z ; m++; &A[m] ) {
    if ( (m % 3) < 2 ) {
        A[m+1] = m*2; // Local
        A[m+10] = m-3; // Remote
    }
    else {
        A[m+1] = m*2; // Remote
        A[m+10] = m-3; // Local
    }
}
```

Privatization results

- update: 128000 elems, blocked by 32, 1000 iter.

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

```
TYPE *lptr = (TYPE*) &A[BF*MYTHREAD];
for(i=BF*MYTHREAD, blk=0; i<N; i+=BF*THREADS, blk++)
    for (j=0; j < BF; j++) lptr[(blk*BF)+j] = j+i;
```

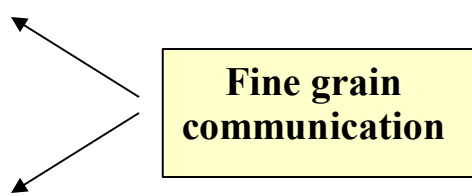


128 Million updates PWR4 SMP

Optimizing share remote accesses

- Parallel loop reshaping allow each thread to access shared local data
- Privatization of local shared accesses reduces the number of runtime calls
- **How do we deal with the remaining shared remote accesses ?**

```
upc forall (m=0; m < Z ; m++; &A[m] ) {  
    if ( (m % BF) < K ) {  
        ... // privatized  
        __xlupc_assign_array(A_h, __t, m+10, ...);  
    } else {  
        ... // privatized  
        __xlupc_assign_array(A_h, __t, m+1, ...);  
    }  
}
```



**Fine grain
communication**

Aggregate fine grained communication

- The idea is to combine multiple remote accesses together to reduce communication cost
 1. Group remote shared references based on their base symbol, the partition they map to and the access type (use, def)
 2. Use local buffers for the bulk transfers
 3. Replace original shared references

shared [BF] int RES[N], V1[N], V2[N];

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

RES[i] = V1[i+BF] + V2[i+BF];

Distance vectors:
 V1: [BF], V2: [BF]
 → Always remote

```
int lv1[BF], lv2[BF];
```

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

```
  if (i%BF== 0) {
    upc_memget(&lv1[0], &V1[i+BF], BF*sizeof(int));
    upc_memget(&lv2[0], &V2[i+BF], BF*sizeof(int));
```

```
    j = 0;
```

```
    RES[i] = lv1[j] + lv2[j];
    j++;
  }
```

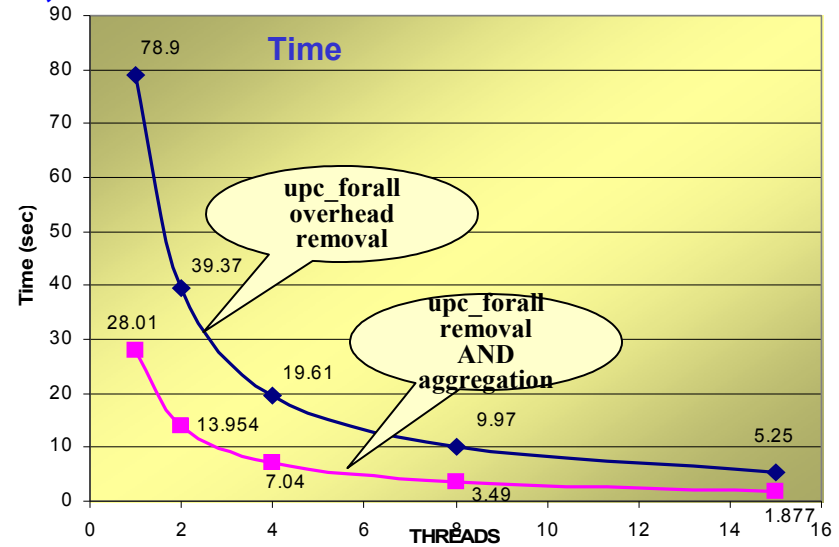
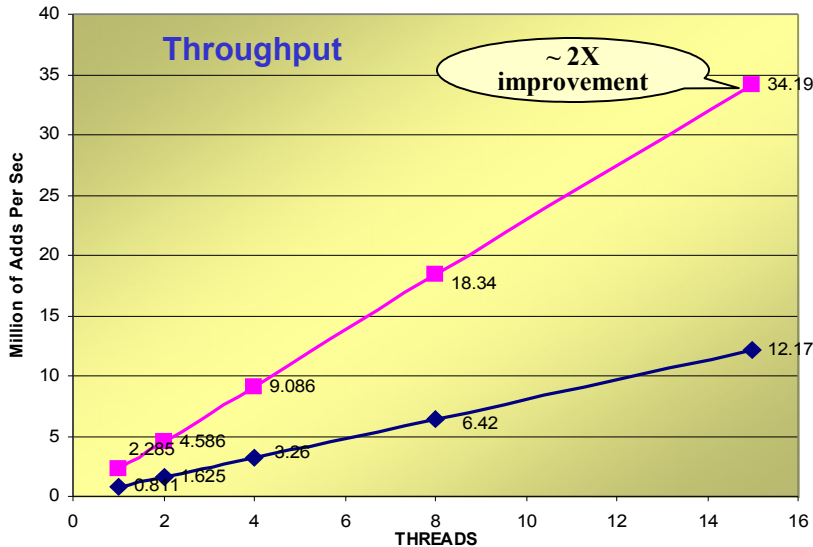
**Bulk transfers each
block of "BF" elements**

Aggregate fine grained communication results

- vector add: 64000 elems, block by 32, 1000 iter.

```
shared [BF] int RES[N], V1[N], V2[N];
upc_forall (i=0; i<N; i++; &RES[i]) {
    RES[i] = V1[i+BF] + V2[i+BF];
}
```

```
upc_forall (i=0; i<N; i++; &RES[i]) {
    if (i%BF== 0) {
        upc_memget(&lv1[0], &V1[i+BF], BF*sizeof(int));
        upc_memget(&lv2[0], &V2[i+BF], BF*sizeof(int));
        j = 0;
    }
    RES[i] = lv1[j] + lv2[j];
    j++;
}
```



Hide the latency of shared accesses

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

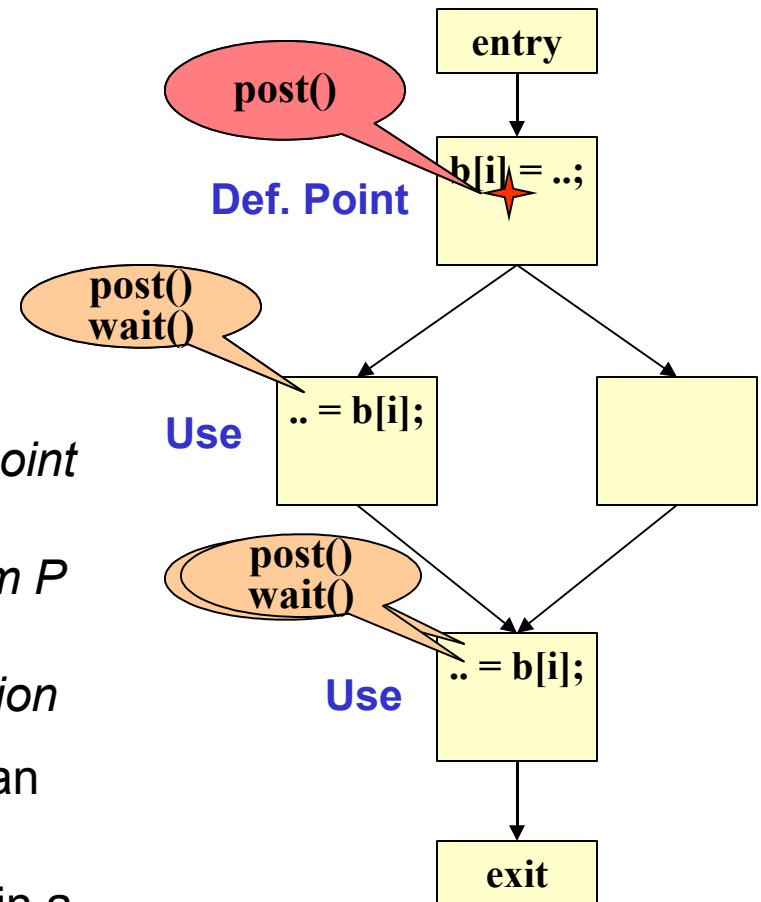
```
__xlupc_deref_array(C_h, __t1, i, sizeof(int), ...); ← wait
__xlupc_deref_array(B_h, __t2, i, sizeof(int), ...); ← wait again
__t3 = __t1 + __t2;
__xlupc_assign_array(A_h, __t3, i, sizeof(int), ...); ← wait yet again !
```

- 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

Scheduling of post/wait calls

GOAL: place the post call as early as possible

- Collect shared loads uses in a sync. region
- 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



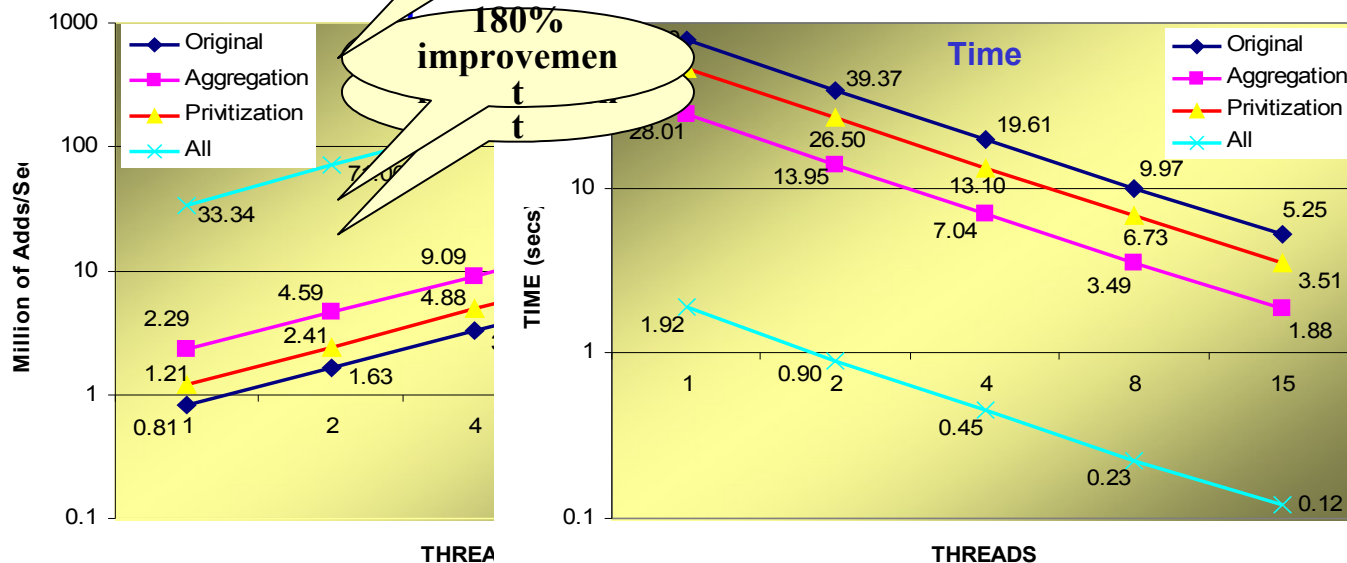
Effect of combining the optimizations

```
shared [BF] int RES[N], V1[N], V2[N];
upc_forall (i=0; i<N; i++; &RES[i]) {
    RES[i] = V1[i+BF] + V2[i+BF];
}
```

```
for (i=BF*MYTHREAD, blk=0; i<N; i+=BF*THREADS, blk++) {
```

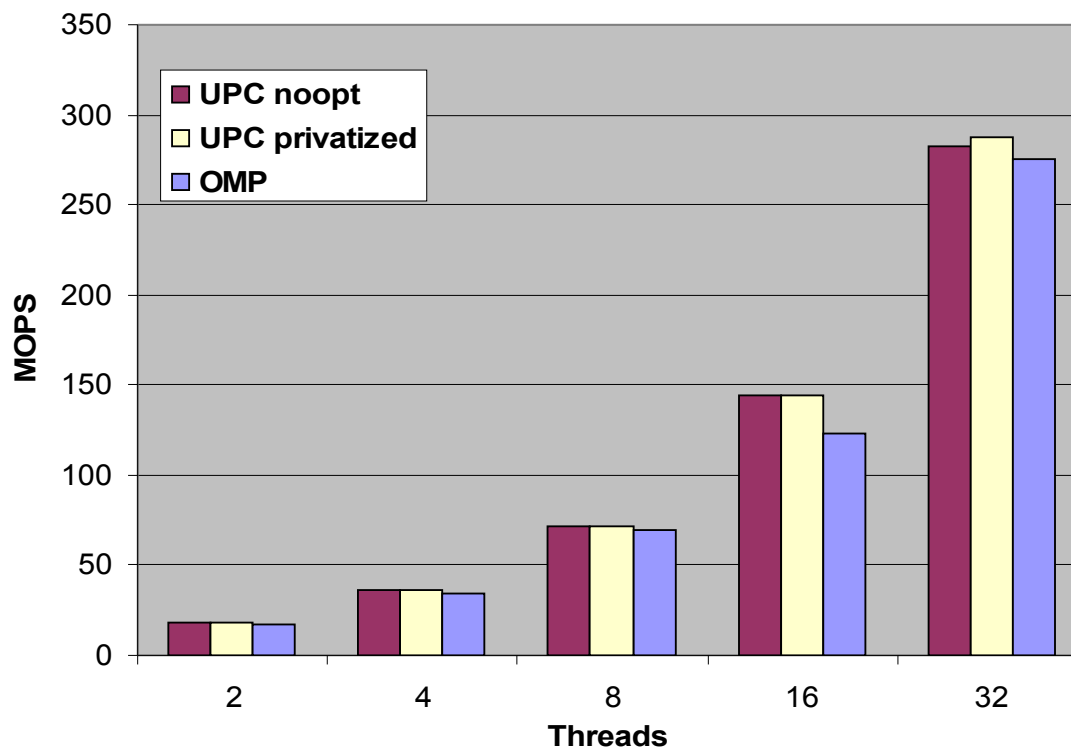
```
    if (i%BF==0) { // aggregate V1 and V2
        upc_memget(&lv1[0], &V1[i+BF], BF*sizeof(int));
        upc_memget(&lv2[0], &V2[i+BF], BF*sizeof(int));
```

```
    }
    ptr[(blk*BF)+j] = lv1[j] + lv2[j];
    ; j++) // privatize RES
```



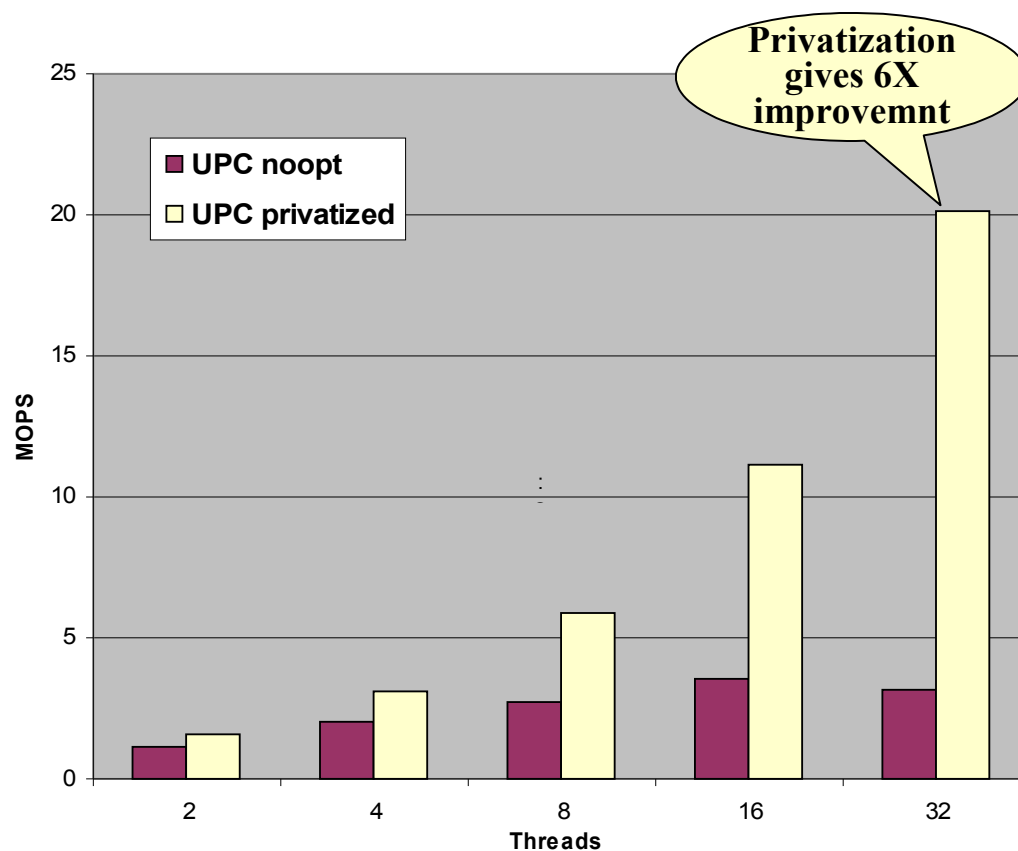
EP Class B – UPC vs OMP

- Very few shared references are privatized
- UPC program performs like the OMP version



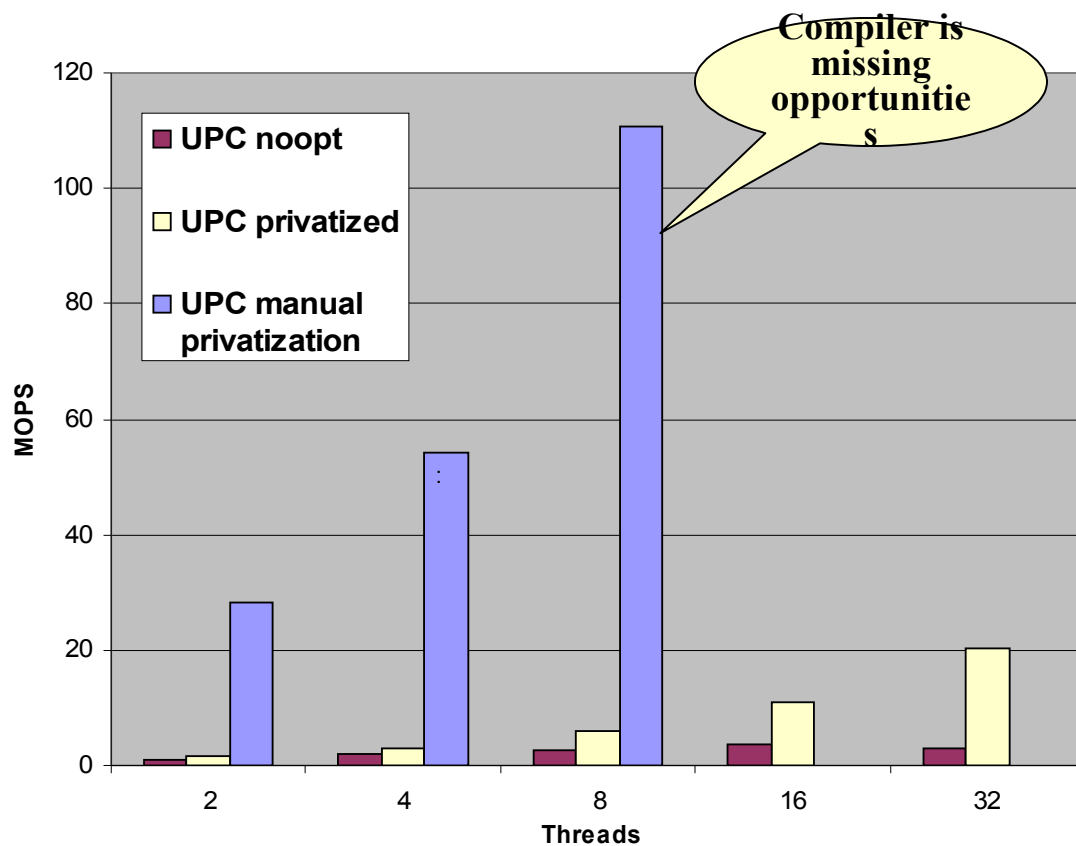
IS Class B: privatization improvement

- Compiler privatized shared references gives a nice improvement



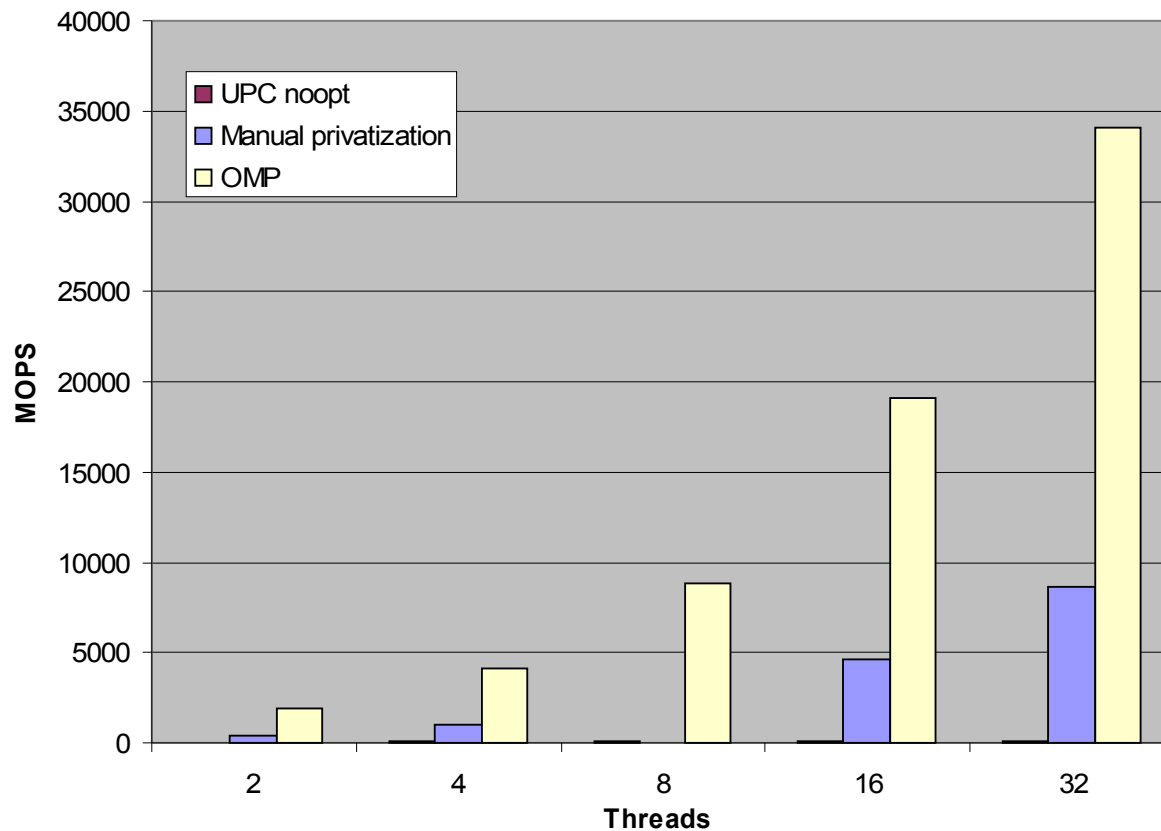
IS Class B: missing opportunities

- Manual privatization and aggregation indicates there are opportunities for the compiler



FT Class A: a case for more UPC optimizations

- Privatization helps but is not sufficient ...
- Aggregation of messages and non-blocking calls ?



UPC Development team

- **IBM Toronto Software Development Lab**
 - Roch Archambault, Raul Silvera, Ettore Tiotto, Philip Luk, Kevin Lou, Raymond Mak, Lawrence Hu, Irina Rada, ...
- **IBM T.J. Watson Research Center**
 - Călin Cașcaval (lead), George Almási, Tony Bolmarcich, ...
- **University of Alberta**
 - Kit Barton, Rahul Garg
- **Technical University of Catalunya (UPC)**
 - Montse Farreras (Myrinet messaging)

AIX SMP XLUPC Alpha available for downloads at:

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