
Communication Optimizations in the Berkeley UPC Compiler

Wei Chen, Costin Iancu, Kathy Yelick
UC Berkeley/LBNL
<http://upc.lbl.gov>

Partitioned Global Address Space

- **PGAS languages share the global address space abstraction**
 - Shared memory is partitioned by processors
 - User controlled data layout (global pointers and distributed arrays)
 - One-sided communication through reads/writes of shared variables (both fine-grained and bulk)
 - Languages: Titanium, Co-Array Fortran, [UPC](#)
- **Growing support from advanced architectures**
 - Hardware RMA in Cray X1E, Cray XD1, SGI Altix, etc.
- **Improves productivity for irregular applications**
 - Fast prototyping of complex algorithms
 - No need to explicitly insert communication into code

Unified Parallel C

- A parallel extension of C
- SPMD parallelism
- Several kinds of shared array distributions
- Pointers to access (possibly remote) shared data

1D Stencil Code in UPC

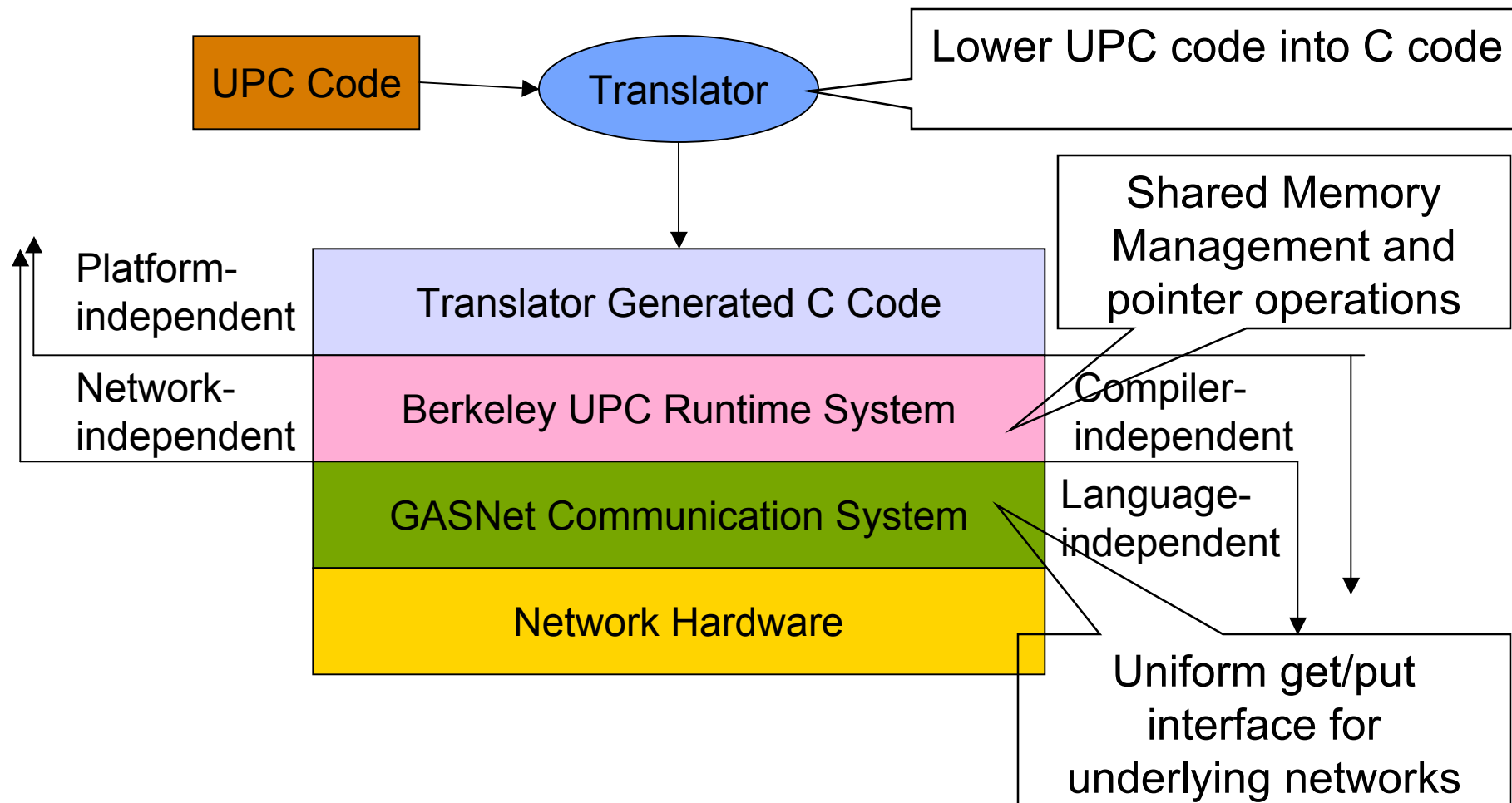
```
shared [LSIZE] double input [LSIZE*THREADS];
shared [LSIZE] double output [LSIZE*THREADS];
upc_forall (j=1; j < LSIZE*THREADS-1; j++; &(input[j])) {
    output[j]=0.25*(2*input[j]+input[j-1]+input[j+1]);
}
```

Motivation

- **Good performance on clusters requires coarse-grained communication**
- **Coarsening requires non-trivial source level transformations**
 - Data layout change, algorithmic change
- **Some operations are much easier to do with fine grained**
 - data setup, directories, dynamic data structures.
- **Programs written in this style:**
 - Communication overhead (3 orders of magnitude over local ld/st)
 - Serial overhead due to shared pointer arithmetic (10x-100x slower than C pointer add)
- **Compiler optimizations required to lower the additional overhead**
 - Ideally, make code “performance portable”.

Overview of the Berkeley UPC Compiler

Two Goals: **Portability** and **High-Performance**



Communication Optimizations for Fine-grained Programs

- **Naïve scheme (blocking call for each load/store) not good enough**
- **PRE on shared expressions**
 - Reduce the amount of unnecessary communication
 - Apply also to UPC shared pointer arithmetic
- **Split-phase communication**
 - Hide communication latency through overlapping
- **Message coalescing**
 - Reduce number of messages to save startup overhead

PRE for Shared Expressions

- **Standard optimization based on Open64's HSSA (Hashed SSA) form**
 - use-def for every variable/indirect load (x , $*p$, $a[i]$, $**q$, etc)
 - Maydef/mayuse to support aliased accesses
- **Hard to reuse existing Open64 PRE infrastructure**
- **Find a single *definition point* for the shared expression e**
 - For each variable/indirect load in e , find its def point (from the HSSA)
 - Pick the point dominated by all others (the one occurring last)
- **Expression can be evaluated just once at the def point**
 - All values used by e are already available
 - All uses are dominated by the single def point

```
shared int * p;  
...  
if (...)  
    i = ...  
if (...)  
    ... *(p+i)  
    ... *(p+i)
```

```
p1 = foo();
```

```
i2 = ...
```

```
i3 = Φ(i2, i1)  
t = p1 + i3
```

*Def point for (p₁+i₃)
(dom all uses)*

```
... *(p1+i3)
```

```
... *(p1+i3)
```


Split-phase Communication

- **Goal:** separate *init* of a get/put as far away as possible from its *sync*
- **Minimize the amount of time waiting on sync**
- **Get: Move init call up**
 - May not move sync down (value is needed right away)
 - But can issue init once the value is available
- **Put: Move sync call down**
 - May not move init up
 - But can delay sync till another conflicting read/write
- **Bulk: look for special patterns**
 - Pipelining all-to-all exchange, etc.
 - Problem is figuring out the conflicts statically

Moving Shared Reads

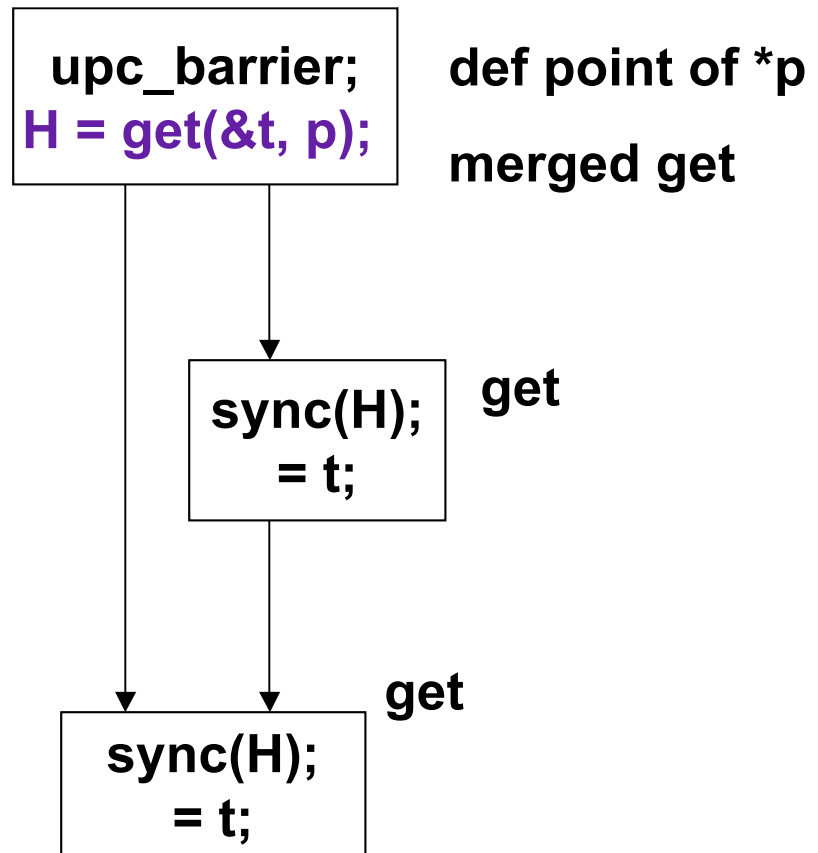
- **Algorithm is same as that of PRE**
 - Put *init* call at *definition point* of the read
 - *Sync* the call right before an use
- **One catch: can not allow speculative load**
 - Doing a load that would not be executed in unoptimized code
 - **Correctness**: loading an incorrect address could segfault
 - **Performance**: even if value not used, still must block for its completion at function exit
- **Runtime support would be nice, but not easily implementable on clusters**

Preventing Speculative Load

- **Every get must be *anticipatable* at the place it's inserted**
 - guaranteed to be executed from there to exit
- **Step 1: Put a get in every basic block containing the load**
- **Step 2: Merge the gets**
 - Bottom-up pass on the flow graph
 - If all of a bb's children have a get, move the get into bb
 - stop at definition point of the expression

Split-phase Reads Example

```
shared double *p;  
...  
upc_barrier;  
if (...)  
    ... *p;  
... *p;
```



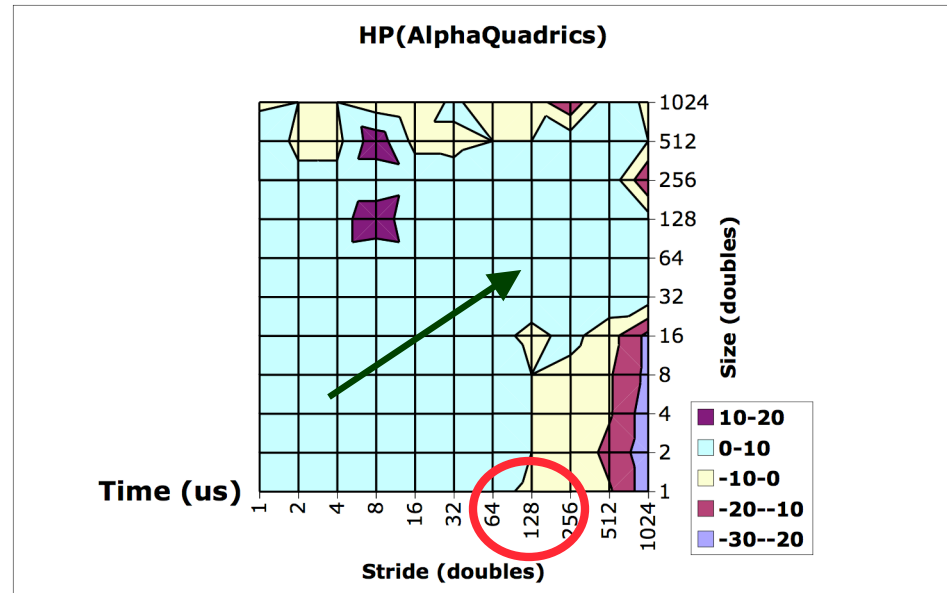
Moving Shared Writes

- HSSA does not give *def-use* information
- Use a path-sensitive algorithm to push down syncs
- Check along every path whether we encounter a statement **s** that uses/modifies the write
 - If yes, insert sync before **s** and terminate the current path
 - Also insert a sync if we reach function exit
 - Minimize the number of redundant syncs
- Each basic block examined at most once (either it has a sync or it doesn't)
- Not as important due to owner computes rule

Coalescing

- **Combine small messages to save startup overhead**
- **Consider get/put already split-phased**
- **Three cases for *get1* and *get2*:**
 - Contiguous: always coalesce
 - Non-contiguous, but with same base address: Bounding box based on performance model (next slide)
 - Unknown: Let runtime decide

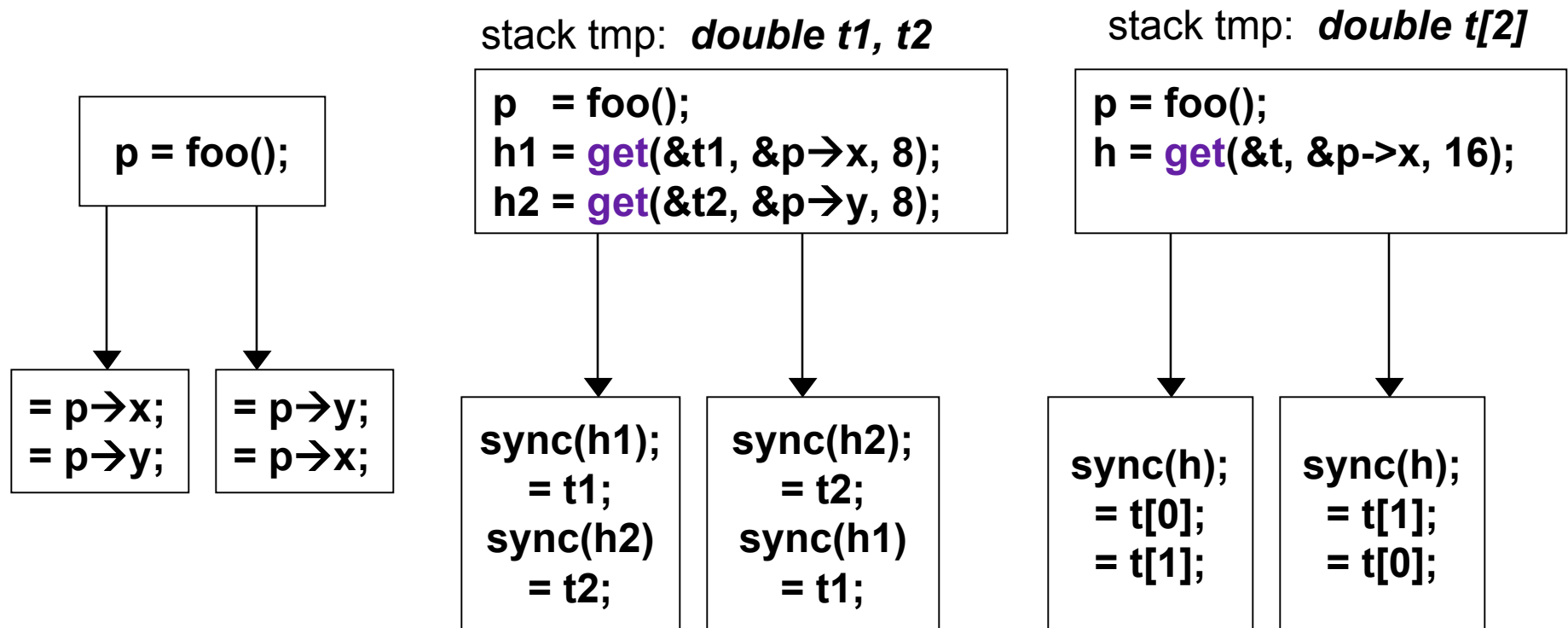
Deciding When to Coalesce



- **Tradeoffs between coalescing (bounding box) and pipelining**
 - Less message vs. less volume
 - with different stride (x-axis) and size (y-axis)
- **1K byte seems to be a good cutoff point**

Coalescing Example

shared struct { double x; double y;} *p;



a). Original UPC code

b). After split-phase

c). After coalescing

Dealing with Consistency Models

- Memory model prevents accesses from being reordered with synchronization constructs
- **upc_barrier**: model as black box that modifies every shared variable
- **Locks: must protect critical section**
 - **upc_lock**: backward code motion barrier
 - **upc_unlock**: forward code motion barrier
- **strict vs. relaxed accesses**
 - Relaxed access behaves like local load/store
 - Strict access is like sequential consistency
 - Model strict accesses as barriers to prevent reordering
 - Conservative, but works in practice

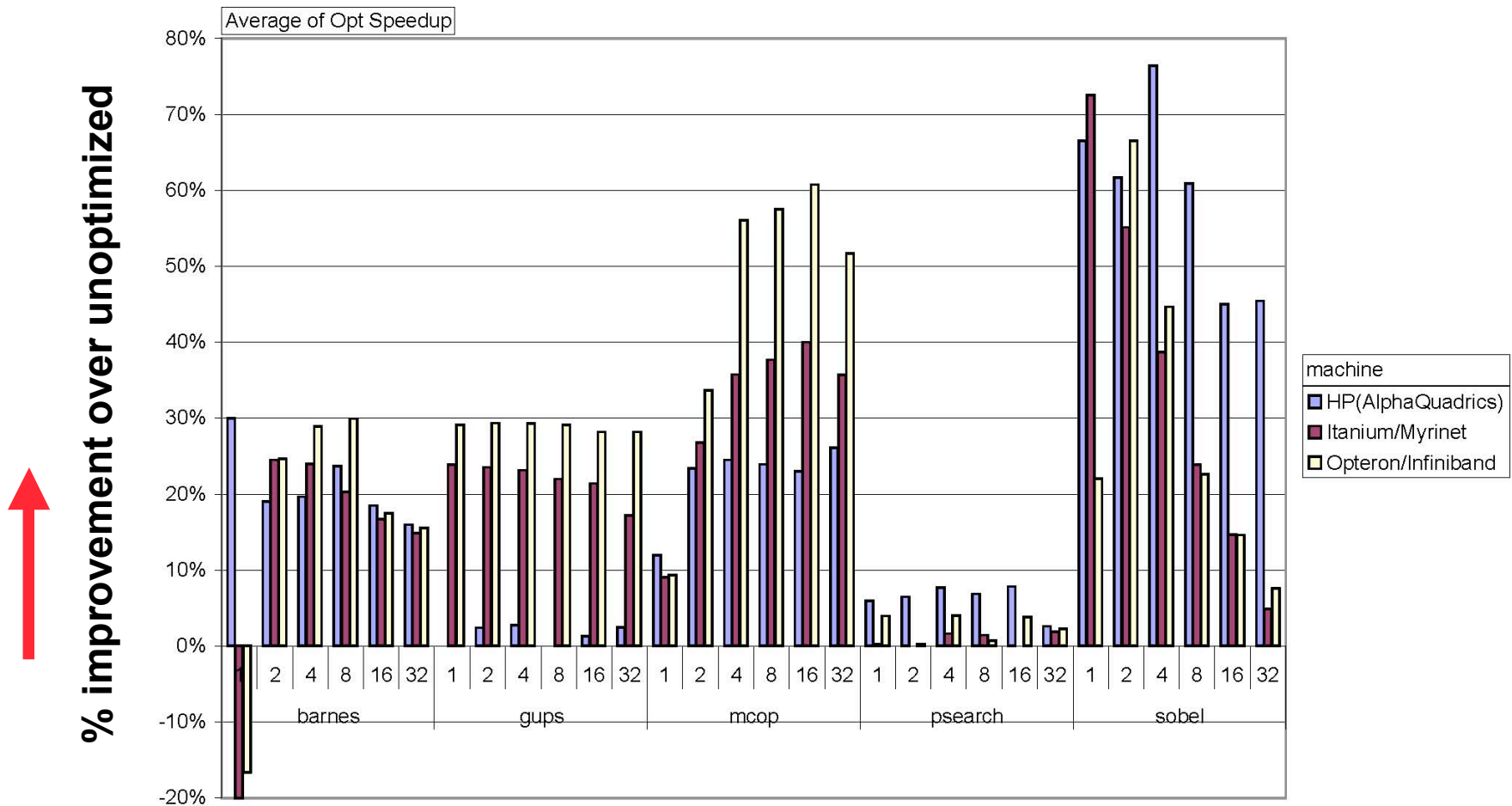
Benchmarks

- **Written by people outside of our group**
- **Gups**
 - Random access (read/modify/write) to distributed array
- **Mcop**
 - Parallel dynamic programming algorithm
- **Sobel**
 - Image filter
- **Psearch**
 - Dynamic load balancing/work stealing
- **Barnes Hut**
 - Shared memory style code from SPLASH2
- **NAS FT/IS**
 - Bulk communication

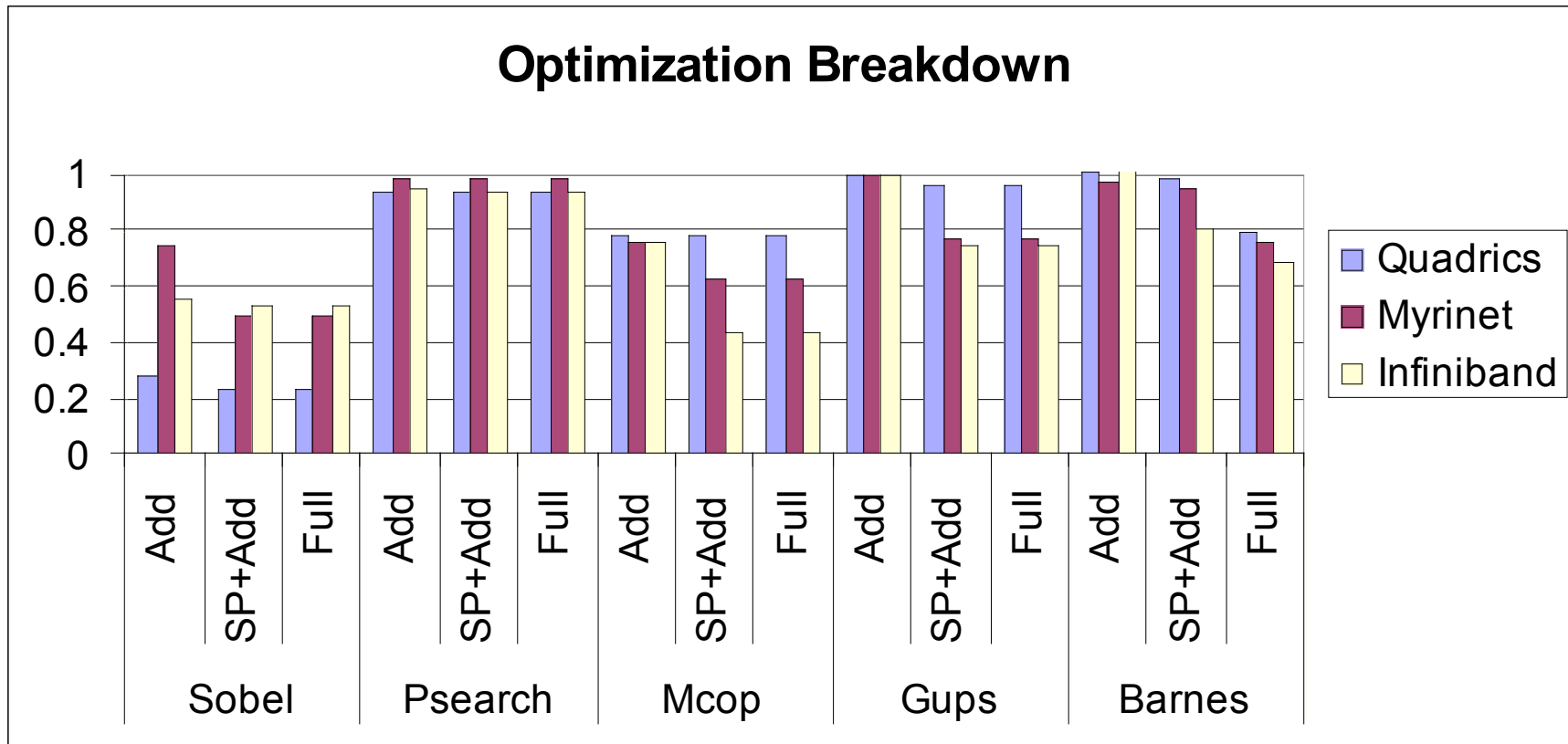
Hardware Testbed

| | Lemieux | RTC | Jacquard |
|------------------|--------------------------|--------------------------|-------------------------------|
| Processor | 1GHz Alpha | 900MHz Itanium2 | 2.2GHz Opteron |
| Network | Quadrics elan3 | Myrinet gm 1.6.5 | Mellanox Infiniband 4x |
| Software | Tru64 gcc 3.4 | Linux gcc 3.4 | Linux gcc 3.3 |
| Get | 7us | 26us | 11.6us |
| Put | 6.5us | 16us | 8.4us |
| Add | 110ns | 220ns | 72ns |

Performance Improvements



Optimization Breakdown



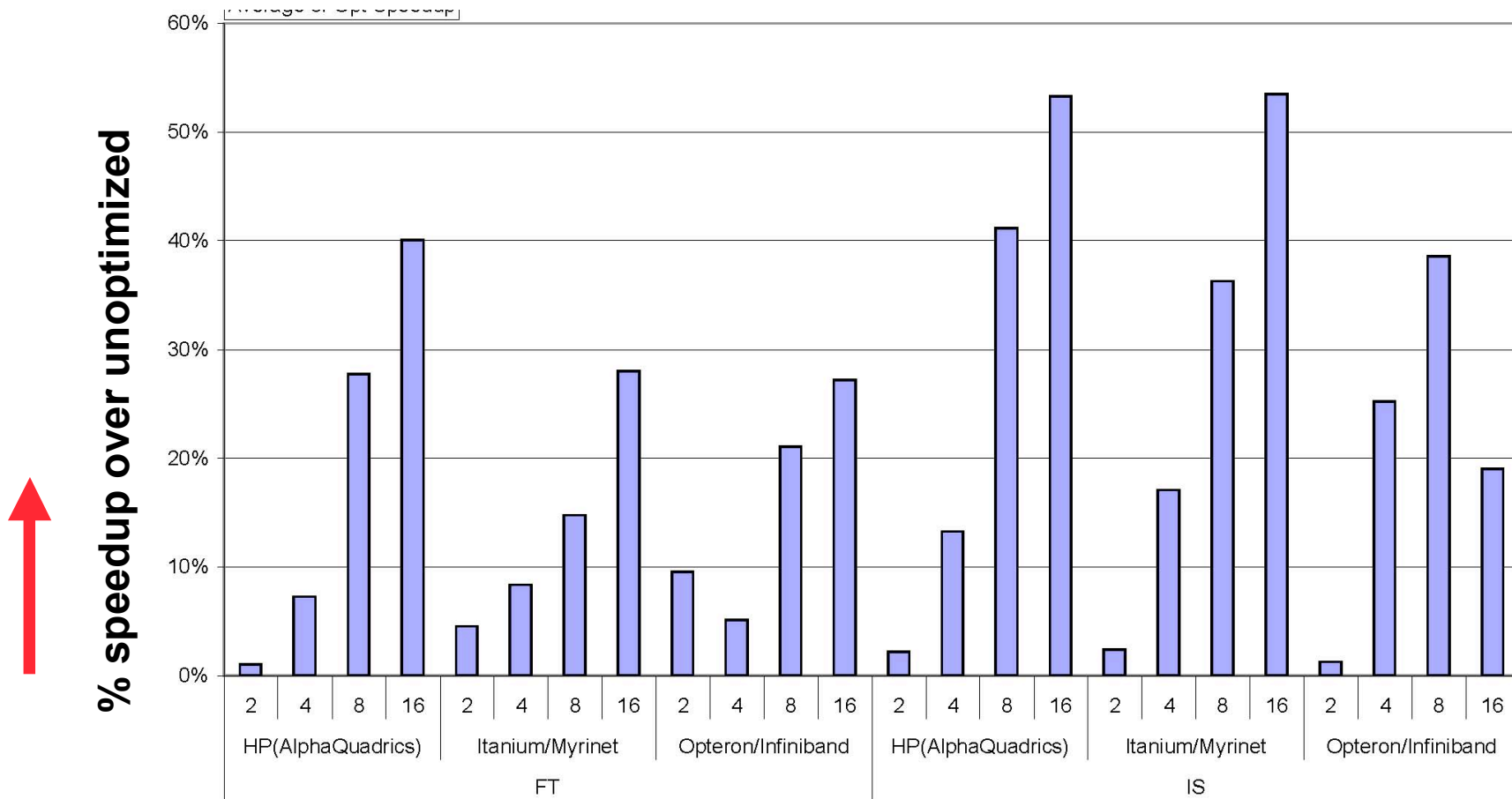
- Compares add-only, split-phase+add, and everything
- Effectiveness of optimization depends on the application
- Effectiveness also depends on networks

Scalability

| Benchmarks | Quadrics | Myrinet | Infiniband |
|------------|-----------|-----------|------------|
| Gups | 29.3 | 22.8 | 39.1 |
| Mcop | < 1 (9.8) | < 1 (10) | < 1 (10.5) |
| Sobel | 46 | 1.3 | 0.53 |
| Psearch | 23 | 6.5 | 16.5 |
| Barnes | < 1 (3.8) | < 1 (2.3) | < 1 (4.3) |

- $T_{base}(1) / T_{opt}(32)$ (32 is linear speedup)
- Parallel speedup in parenthesis -- $T_{base}(2)/T_{opt}(32)$
- Data locality is important
 - Some programs don't scale due to inefficient data layout
 - e.g., blocking the mcop code results in near linear speedup

Performance Improvement for Bulk Code



- Optimizes the all-to-all exchange in FT and IS

Conclusions

- **Our optimizations are effective at reducing communication overhead for fine-grained programs**
- **Global communication scheduling does not seem to improve performance**
- **Relaxed memory models needed to enable optimizations**
- **For scalability, programmers needs to be data locality aware**
- **Download our compiler at <http://upc.lbl.gov>**

Questions?