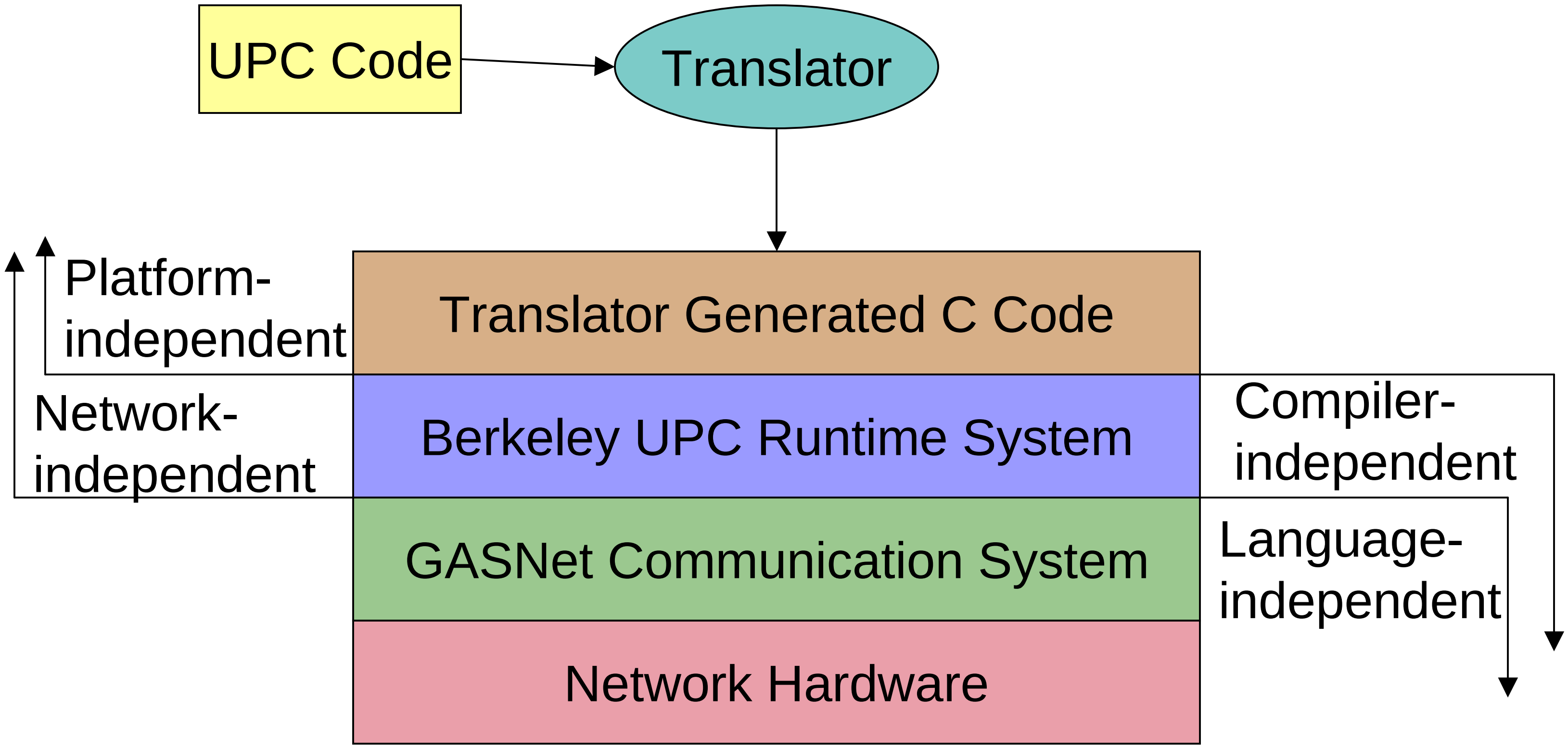
Compiler Optimizations in the Berkeley UPC Translator

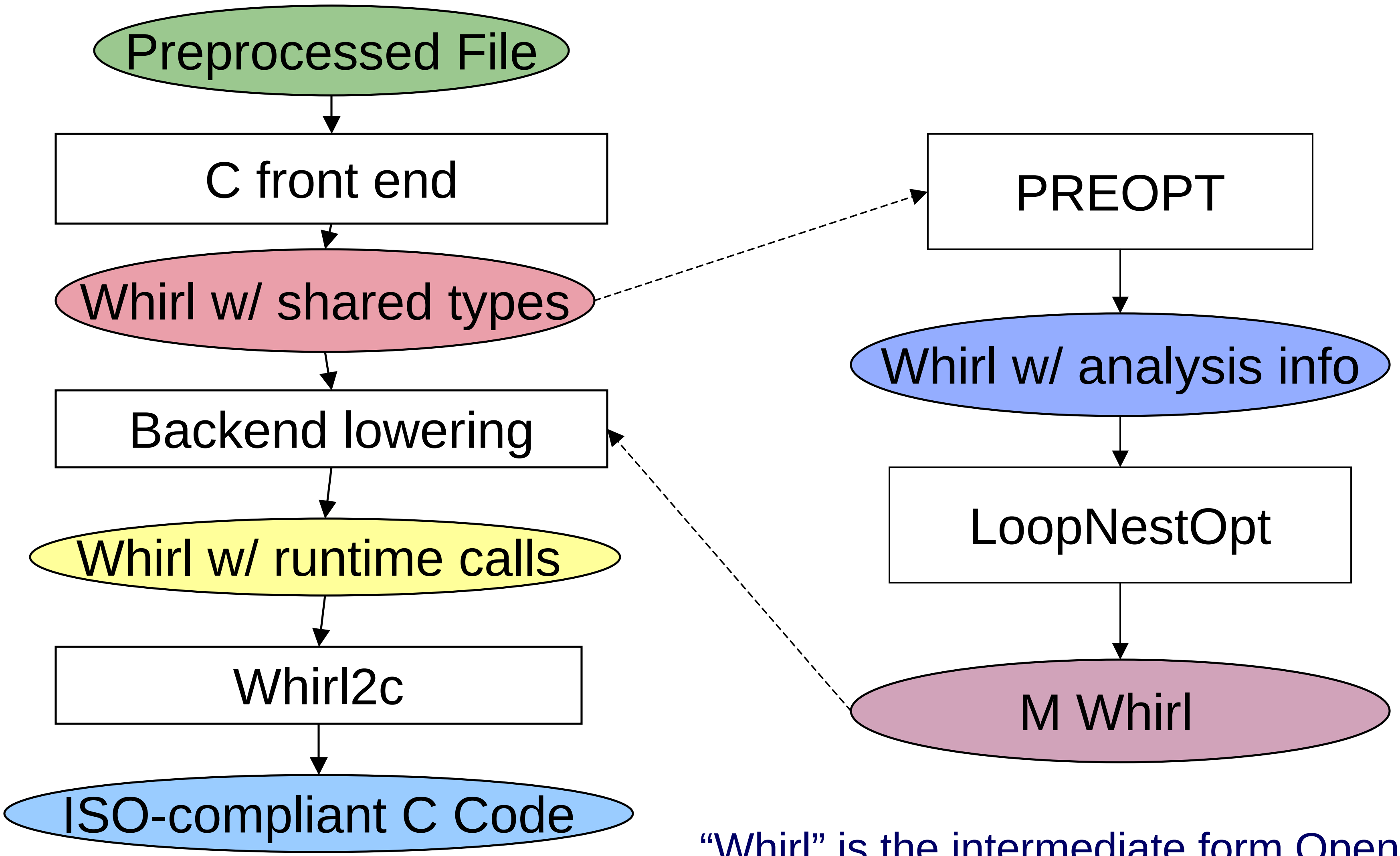
Wei Chen
the Berkeley UPC Group

Overview of Berkeley UPC Compiler



Two Goals: Portability and High-Performance

UPC Translation Process



"Whirl" is the intermediate form Open64

Preopt Phase

- Enabling other optimization phases
  - Loop Nest Optimization (LNO)
  - Whirl Optimizations (WOPT)
- Cleanup the control flows
  - Eliminate gotos (convert to loops, ifs)
  - Setup CFG, Def-Use chain, SSA
  - Intraprocedural alias analysis
  - Identifies DO_LOOPS (includes forall loops)
  - Perform high level optimizations (next slide)
  - Convert CFG back to whirl
  - Rerun alias analysis

Optimizations in PREOPT

- In their order of application:
  - Dead store elimination
  - Induction variable recognition
  - Global value numbering
  - Copy propagation (multiple pass)
  - Simplify boolean expression
  - Dead code elimination (multiple pass)

- Lots of effort in teaching optimizer to work with UPC code
  - Preserve casts involving shared types
  - Patch the high-level types for whirl2c use
  - Convert shared pointer arithmetic into array accesses
  - Various bug fixes

Loop Nest Optimizer (LNO)

- Operates on H whirl
  - Has structured control flow, arrays
- Intraprocedural
- Converts pointer expression into 1D array accesses
- Optimizes DO_LOOP nodes
  - single index variable
  - integer comparison end condition
  - invariant loop increment
  - No function calls/break/continue in loop body

Loop Optimizations

- Separate representation from preopt
  - Access vectors
  - Array dependence graphs
  - Dependence vectors
  - Region for array accesses
  - Cache model for tiling loops, changing loop order
- Long list of optimizations
  - Unroll, interchange, fission/fusion, tiling, parallelization, prefetching, etc.
  - May need performance model for distributed environment

Motivation for the Optimizer

- Translator optimizations necessary to improve UPC performance
  - Backend C compiler cannot optimize communication code
  - One step closer to user program
- Goal is to extend the code base to build UPC-specific optimizations/analysis
  - PRE on shared pointer arithmetic/casts
  - Communication scheduling
  - Forall loop optimizations
  - Message Coalescing

Message Coalescing

- Implemented in a number of parallel Fortran compilers (e.g., HPF, F90)
- Idea: replace individual puts/gets with bulk calls to move remote data to a private buffer
- Targets memget/memput interface, as well as the new UPC runtime memory copy library functions
- Goal is to speed up shared memory style code

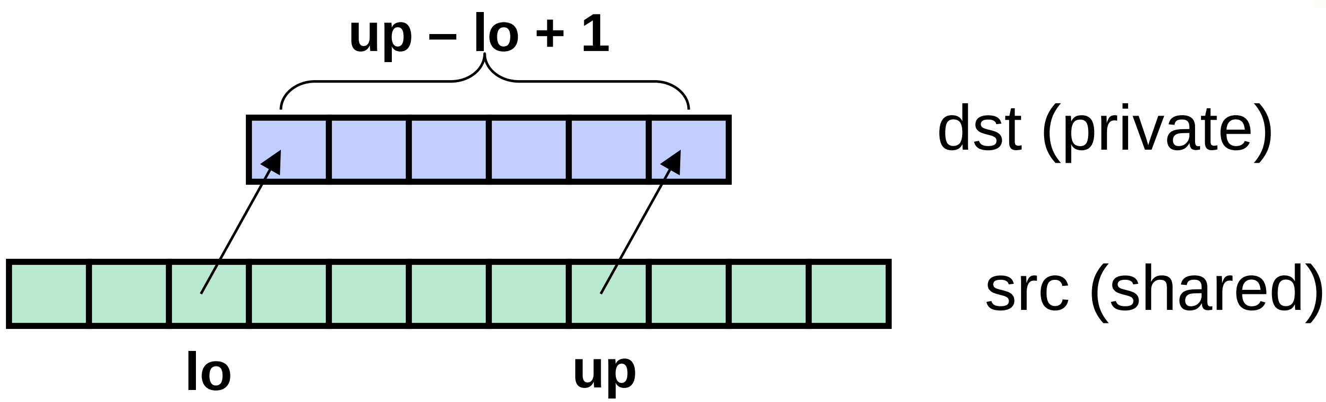
Unoptimized loop	Optimized Loop

`shared [0] int * r;`
`…`
`for (i = L; i < U; i++)`
`  exp1 = exp2 + r[i];`
	`int lr[U-L];`
	`…`
	`upcr_memget(lr, &r[L], U-L);`
	`for (i = L; i < U; i++)`
	`  exp1 = exp2 + lr[i-L];`

Analysis for Coalescing

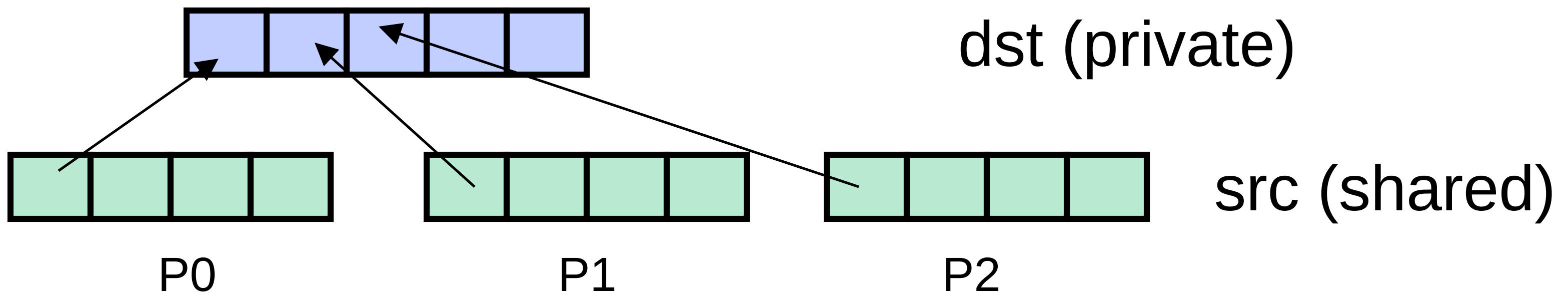
- Handles multiple loop nests
- For each array referenced in the loop:
  - Compute a bounding box (lo, up) of its index value
  - Handles multiple accesses to the same array (e.g., ar[i] and ar[i+1] get same (lo, up) pair)
  - Loop bounds must be loop-invariant
  - Indices must be affine
  - No "bad" memory accesses (pointers)
  - Catch for strict access / synchronization ops in loop – reordering is illegal
- Current limitations:
  - Bounds cannot have field accesses
    - e.g., a + b ok, but not a.x
  - Base address either pointer or array variable
    - No array of structs, array fields in structs

Basic Case: 1D Indefinite Array



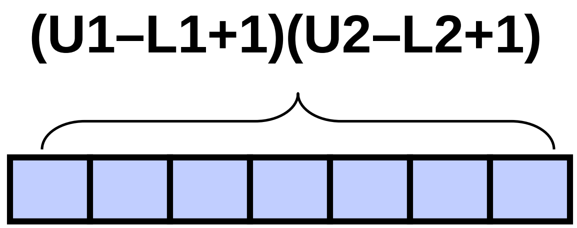
- Use memget to fetch contiguous elements from source thread
- Change shared array accesses into private ones
  - with index translation if (lo != 0)
- Unit-stride writes are coalesced the same as reads, except that memput() is called at loop exit

Coalescing Block Cyclic Arrays

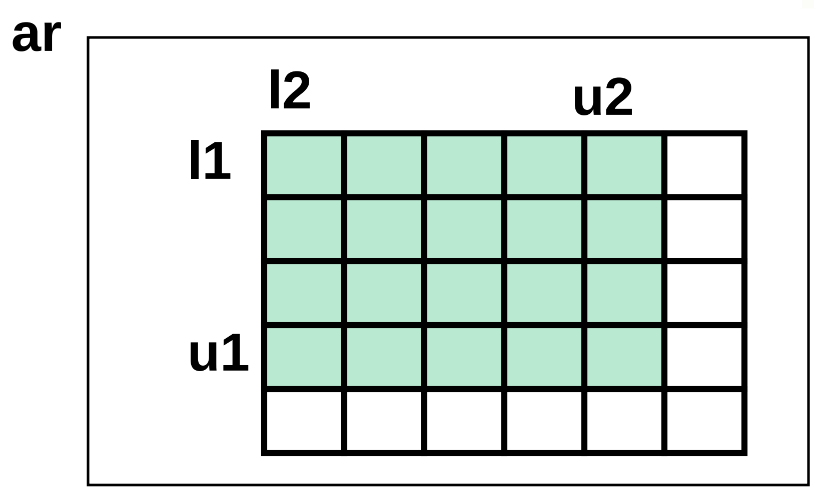


- **May need communication with multiple threads**
- **Call memgets on individual threads to get contiguous data**
  - **Copy in units of blocks to simplify the math**
    - **No. blks per thread = ceil(total_blk / THREADS)**
  - **Temporary buffer: *dst_tmp*[threads][blk_per_thread]**
  - **Overlapped memgets to fill *dst_tmp* from each thread**
  - **Pack content of *dst_tmp* into the *dst* array, following shared pointer arithmetic rule:**
    - **first block of T0, second block of T1, and so on**

Coalescing 2D Indefinite Arrays

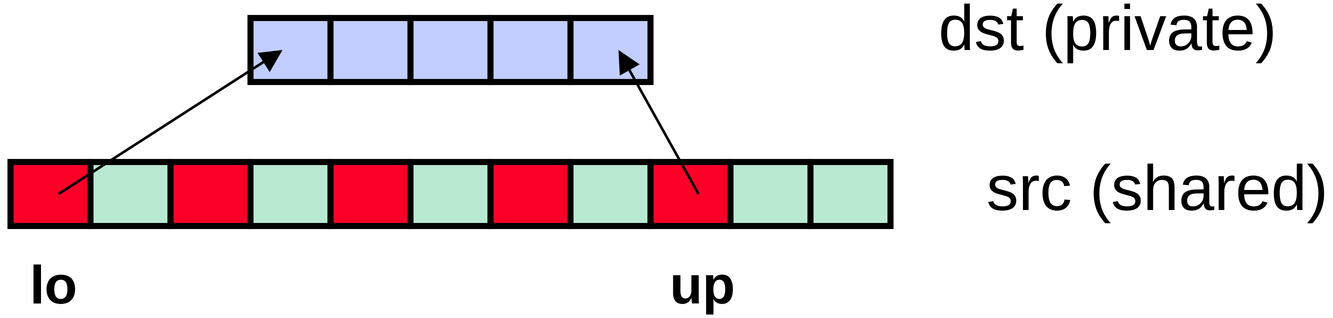



```
for (i = L1; I <=U1; i++)
  for (j = L2; j <= U2; j++)
    exp = ar[i][j];
```



- Fetch the entire rectangular box at once
- Use upc_memget_fstrided(), which takes address, stride, length of source and destination
- Alternative scheme:
  - Optimize the inner loop by fetching one row at a time
  - Pipeline the outer loop to overlap the memgets on each row

Handling Strided Accesses

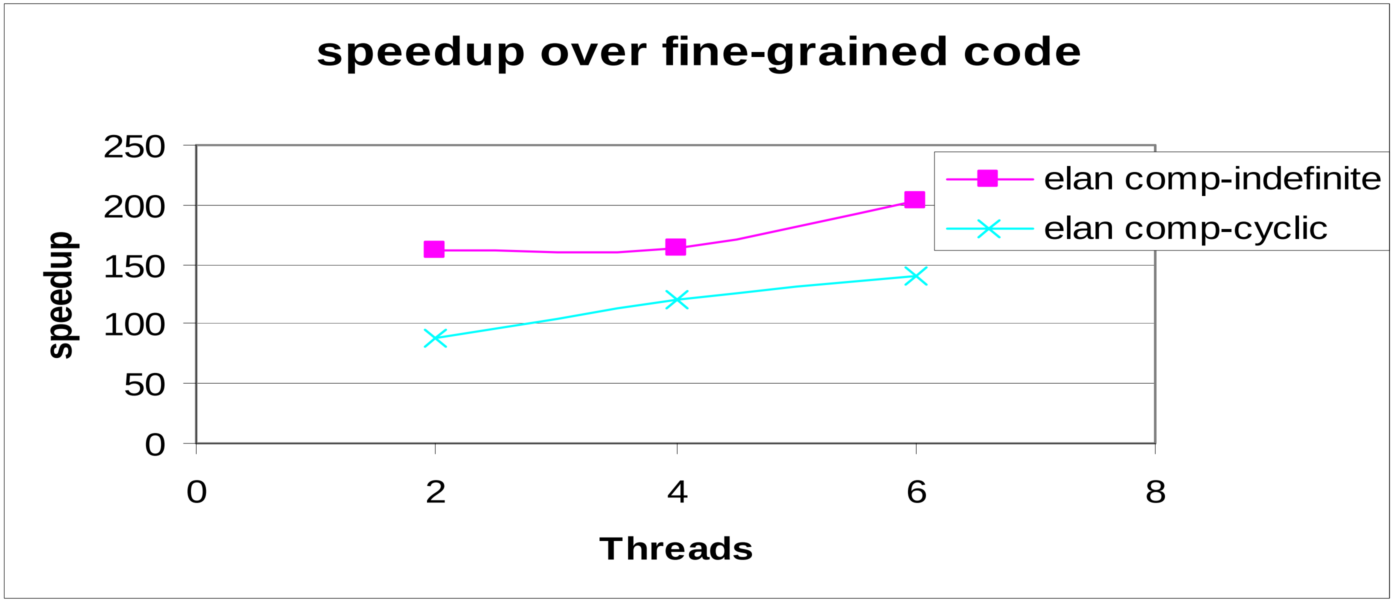


- **Want to avoid fetching unused elements**
- **Indefinite array:**
  - **A special case for *upc_memget_fstrided***
- **Block cyclic array:**
  - **Use the *upc_memget_ilist* interface**
  - **Send a list of fix-sized (in this case 1 element) regions to the remote threads**
  - **Alternatively, use strided memcpy function on each thread**
    - **messy pointer arithmetic, but maybe faster**

Preliminary Results -- Performance

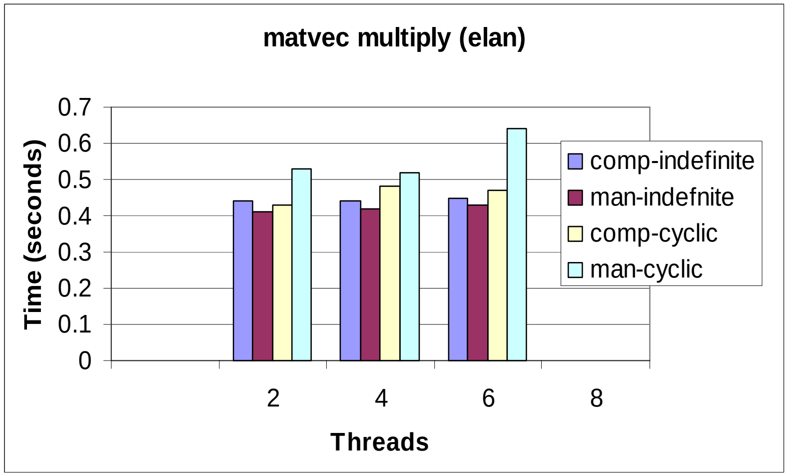
- Use a simple parallel matrix-vector multiply
- Row distributed cyclically
- Two configurations for the vector
  - Indefinite array (on thread 0)
  - Cyclic layout
- Compare performance of the three setup
  - Naïve fine-grained accesses
  - Message coalesced output
  - Bulk style code
    - indefinite: call upc_memget before outer loop
    - cyclic: like message coalesced code, except read from the 2D tmp array directly (avoids the flattening of the 2D array)

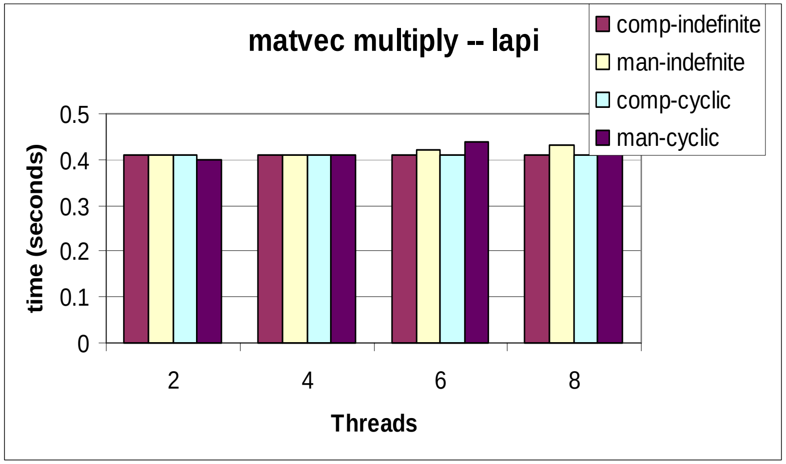
Message Coalescing vs. Fine-grained



- **One thread per node**
- **Vector is 100K elements, number of rows is 100*threads**
- **Message coalesced code more than 100X faster**
- **Fine-grained code also does not scale well**
  - **Network contention**

Message Coalescing vs. Bulk





- **Message coalescing and bulk style code have comparable performance**
  - **For indefinite array the generated code is identical**
  - **For cyclic array, coalescing is faster than manual bulk code on elan**
    - **memgets to each thread are overlapped**

Preliminary Results -- Programmability

- Evaluation Methodology
  - Count number of loops that can be coalesced
  - Count number of memgets that can be coalesced if converted to fine-grained loops
  - Use the NAS UPC benchmarks
- MG (Berkeley):
  - 4/4 memget can be converted to loops that can be coalesced
- CG (UMD):
  - 2 fine-grained loops copying the contents of cyclic arrays locally can be coalesced
- FT (GWU):
  - One loop broadcasting elements of a cyclic array can be coalesced
- IS (GWU):
  - 3/3 memgets can be coalesced if converted to loop

Conclusion

- Message coalescing can be a big win in programmability
  - Can offer comparable performance to bulk style code
  - Great for shared memory style code
- Many choices of code generation
  - Bounding box vs. strided vs. variable-size
  - Performance is platform dependent
- Lots of research/future work can be done in this area
  - Construct a performance model
  - Handling more complicated access patterns
  - Add support for arrays in structs
  - Optimize for special cases (e.g. constant bound/strides)