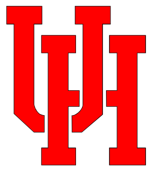
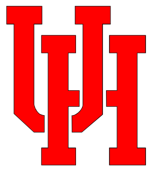


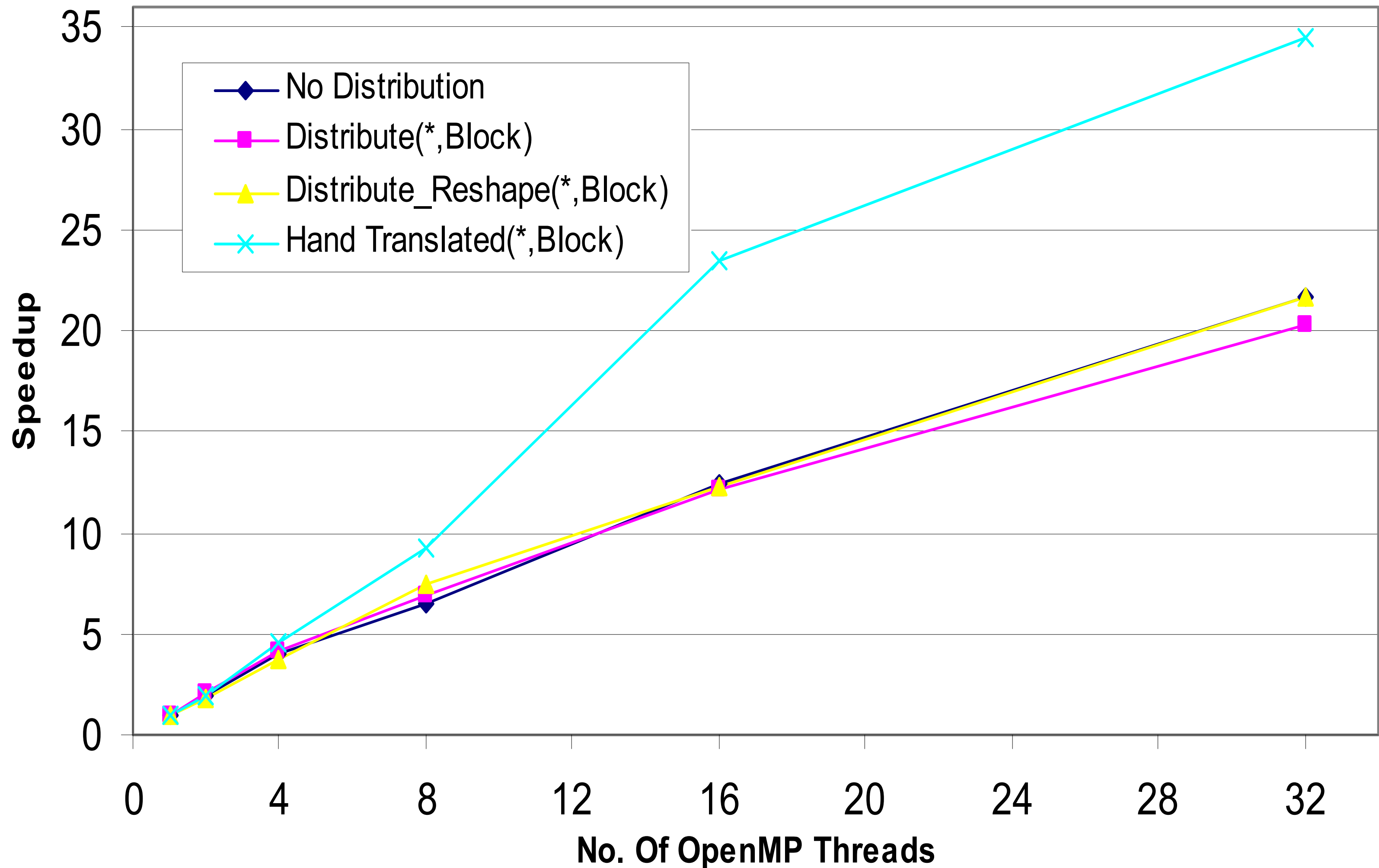
Scalable OpenMP

- CPUs are now comparatively cheap
  - Human labor is not
  - Industry needs a high-level programming model
- OpenMP is widely supported, used for SMP, ccNUMA platforms
- However….
  - It is not easy to create scalable OpenMP code
  - OpenMP requires a shared-address space, but (distributed memory) clusters are in wide use
  - Academia is working on alternative APIs for scalable parallel programming

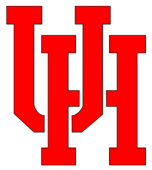


OpenMP Performance

- Performance issues that users must deal with on existing shared memory machines:
  - Cache, esp. false sharing of cached data
  - Minimizing synchronization between threads
  - Balancing workload
  - Ensuring good data locality
- Application developers have obtained scalable codes (up to a point)

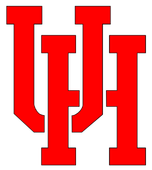


LBE on SGI Origin2000 (256x256)
No Distribution
Distribute(*,Block)
Distribute_Reshape(*,Block)
Hand Translated(*,Block)
Speedup
35
30
25
20
15
10
5
0
0
4
8
12
16
20
24
28
32
No. Of OpenMP Threads



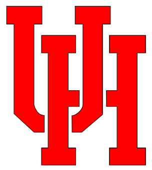
Scalable OpenMP

- Hard manual work required to get good performance
- Compiler alone cannot bridge the gap
- So careful choice of language extensions needed if user is not to do everything
- Good runtime library essential, especially for clusters
  - Collective communications, reductions, synchronization



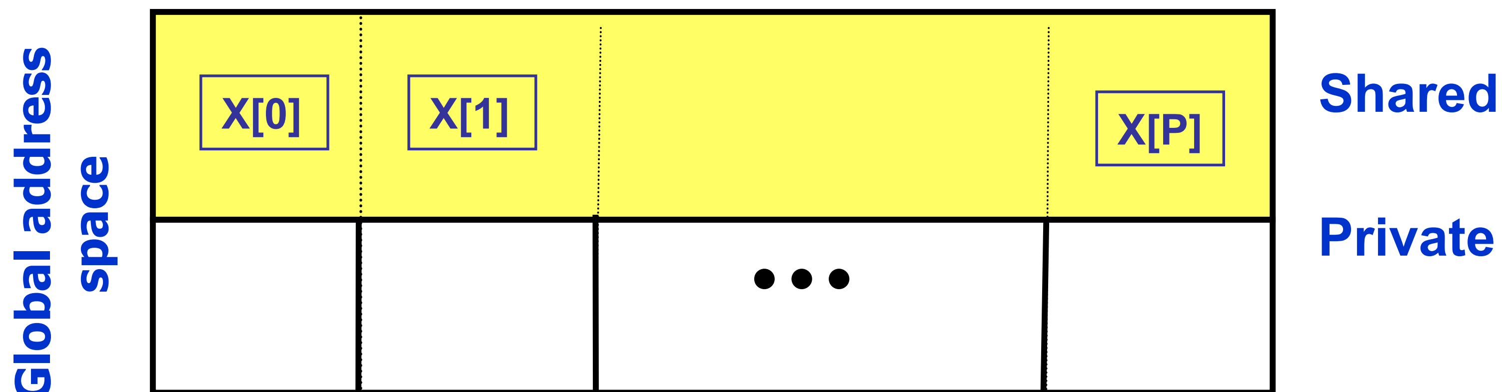
Is it Worth It?

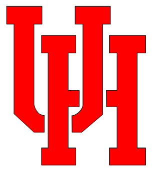
- What might future hardware look like?
  - Massive parallelism
  - Global address space
  - High bandwidth to memory
  - Thread-based
  - Efficient synchronization, context switching
- What features might a new execution model have?
  - Multi-level parallelism, multithreading



Unified Parallel C (UPC)

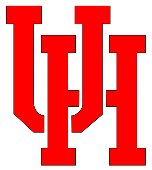
- UPC is an explicitly parallel global address space language with SPMD parallelism
  - An extension of C (CAF is roughly equivalent for Fortran)
  - Shared memory is partitioned by threads
  - One-sided (bulk and fine-grained) communication through reads/writes of shared variables





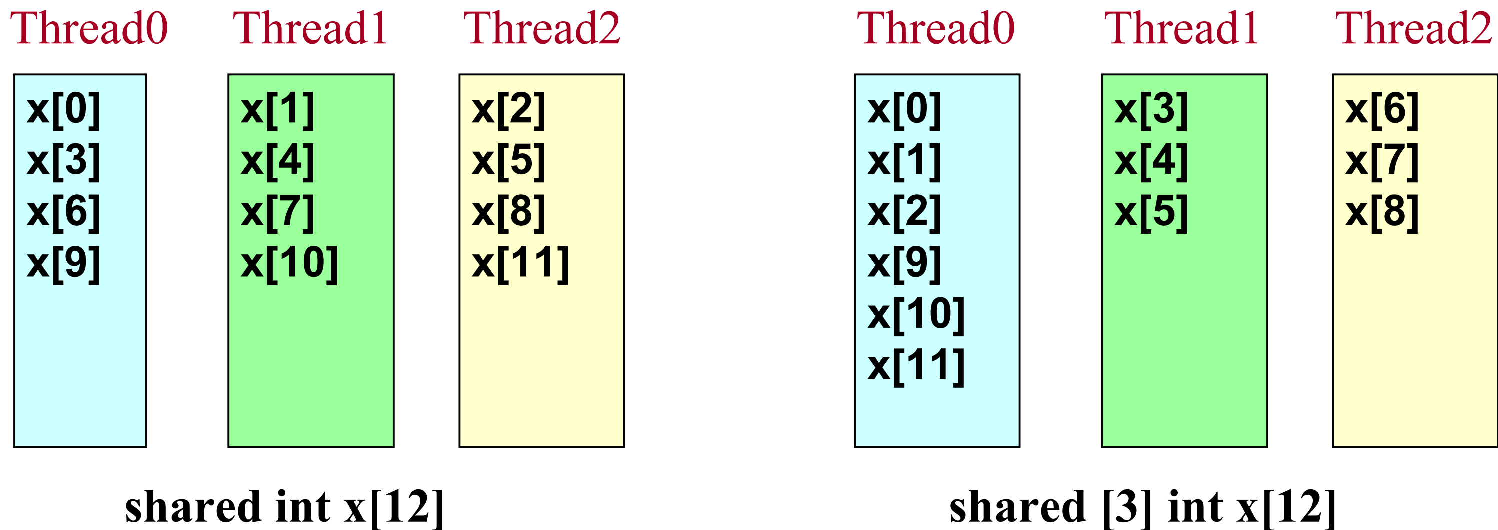
UPC Programming Model

- SPMD parallelism
  - fixed number of images execute asynchronously
- Several kinds of array distributions
  - **double a[n]** private n-element array on each processor
  - **shared double a[n]** n-element shared, cyclic mapping
  - **shared [4] double a[n]** a block cyclic array with 4-element blocks
  - **shared [0] double *a = (shared [0] double *) upc_alloc(n);** a shared array with all elements local
- Pointers for irregular data structures
  - **shared double *sp** a pointer to shared data
  - **double *lp** pointers to private data



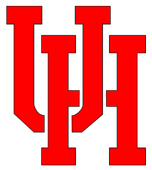
Data Distribution

- UPC distributes shared arrays by blocks across the different threads
  – Shared [block_size] array [number-of-elements]



a. Default block factor

b. User defined blocking factor



Parallel Loops in UPC

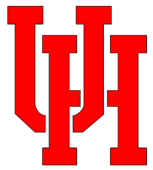
- UPC has a “forall” construct for distributing computation

  Ex: Vector Addition

```
shared int v1[N], v2[N], v3[N];
upc_forall (i=0; i < N; i++; &v3[i]) {
    v3[i] = v2[i] + v1[i];        }
```

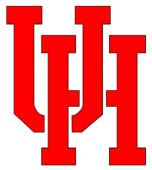
Affinity Exp

- Two kinds of affinity expressions:
  - Integer (compare with thread id)
  - Shared address (check the affinity of address)
- Affinity tests are performed on every iteration



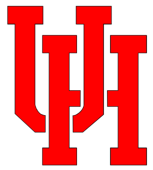
UPC Programming Model Features

- Global synchronization
  - **upc_barrier** traditional barrier
  - **upc_notify/upc_wait** split-phase global sync.
- Pair-wise synchronization
  - **upc_lock/upc_unlock** traditional locks
- Memory consistence has two types of accesses
  - **Strict:** must be performed immediately, atomically: typically a blocking round-trip message if remote
  - **Relaxed:** still must preserve dependencies, but other processors may view these as happening out of order
- Parallel I/O based on ideas in MPI I/O



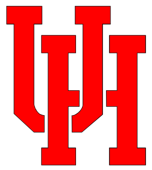
Review of UPC

- Identifies and supports optimization of accesses to non-local data:
  - overcome latencies (overlap computation and communication) via split-phase barriers
  - possible placement analysis, by separating get(), put() as far as possible from sync()
- No implicit synchronization
- Performance on large-scale codes?



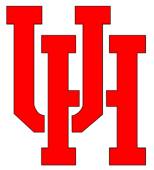
A Few Ideas from Emerging Programming Models

- More general memory consistency models
  - Relaxed consistency models can unleash more parallelism
- Multiple (more) levels of parallelism
- Move program to data instead of data to program
- Reduce cost of sychronization
  - Wait (data)
- Emphasis on programmability



Data Mappings?

- Data mapping only makes sense if it is coupled with a corresponding schedule (for parallel loops)
  - Page mappings? Element mappings?
- But could be very confusing: OpenMP has a work distribution
  - Thus data referenced by each thread is determined by schedule (in parallel loops)
  - Maybe it makes sense to give (only) a data related schedule in OpenMP
  - Best practice?



Some Conclusions

Language must better help user to obtain scalable code

- Encourage more privatization of data
- Reduce synchronization cost
- Write code with hierarchical parallelism

- Can we make it easier to write SPMD code?