COMP 633 - Parallel Computing

Lecture 22
November 17, 2021

Partitioned Global Address Spaces

Parallel languages for distributed memory machines

Topics

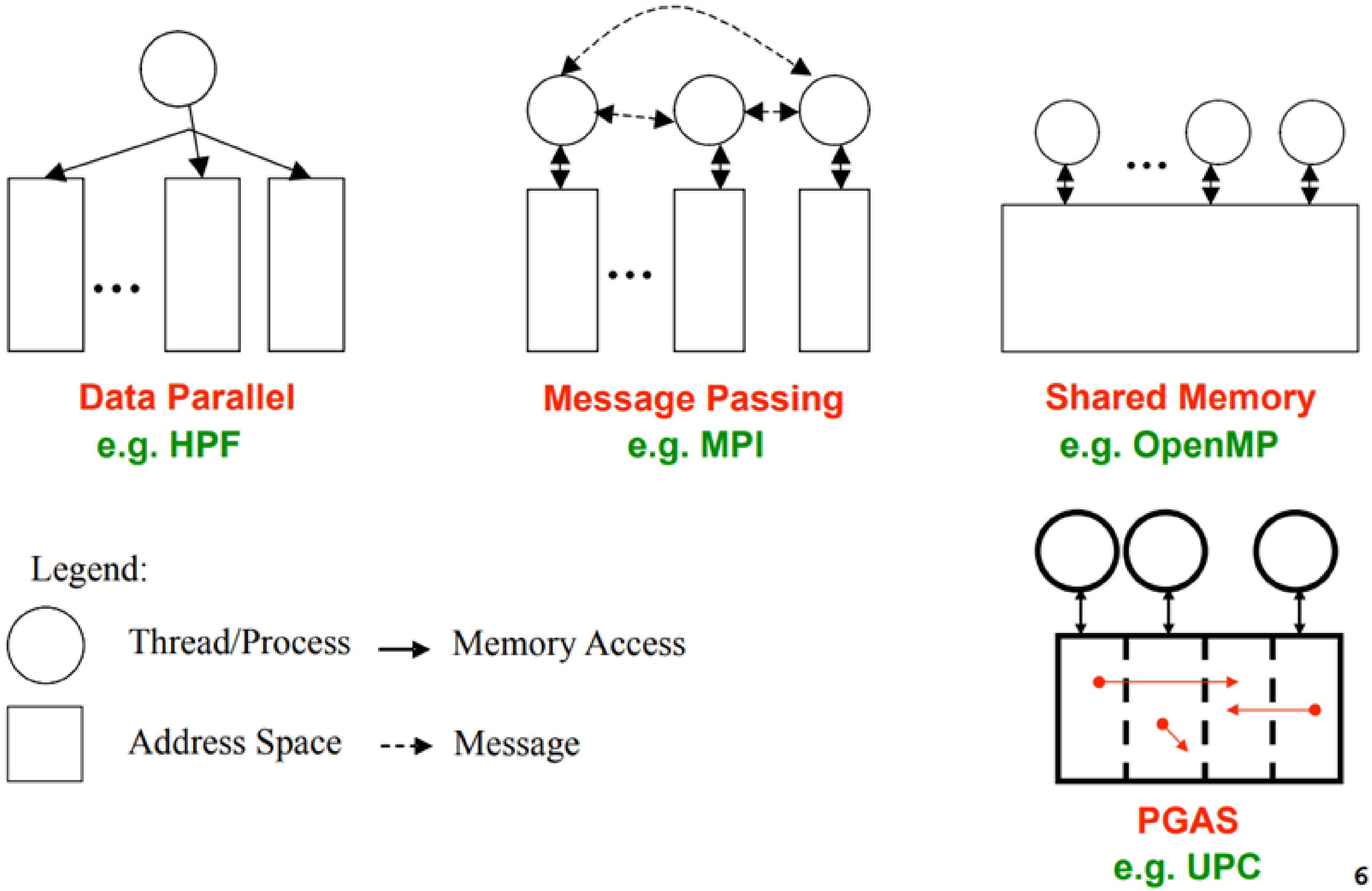
- MPI-2 and MPI-3 specifications for clusters
  - add single-sided communication via remote direct memory access (RDMA)

- High-level parallel programming languages for clusters using RDMA
  - High Performance Fortran (HPF)
  - Unified Parallel C (UPC)

Parallel programming models (thus far)

- Address-space
  - single (shared memory)
  - multiple (distributed memory)
- Source of parallelism
  - SPMD (processor-centric)
  - data parallelism (data-centric)
  - task parallelism (problem-centric)
- Type of synchronization
  - statement-level (SIMD)
  - barrier (SPMD)
  - fork-join (taskwait)
  - mutual exclusion (locks)
  - conditions (signal/wait)
- Inter-processor communication
  - shared memory
  - message passing
    - matching send & receive
  - collective communication
    - broadcasts, reductions
    - gather, scatter and total-exchange
- Memory models
  - distributed memory
    - BSP
    - C + MPI
  - shared memory
    - WT, PRAM
    - Cilk
    - C + OpenMP
    - Java, C + Pthreads

Diagrammatic view of parallel programming models



Remote Direct Memory Access (RDMA)

- Hardware-supported feature of modern cluster interconnects
  - processes can directly read and write memory in other processors
  - in principle, can emulate a global shared memory
    - but remote memory references are slow (mostly communication latency)
    - and have non-uniform access cost (depends on network)

- MPI-2 (sort of) and MPI-3 (more so) introduce one-sided communication operations using RDMA
  - communication (put/get)
  - atomic operations (e.g. atomic add)
  - synchronization
    - recall memory consistency models

- Look at two parallel programming level languages that use RDMA
  - High Performance Fortran (HPF), Unified Parallel C (UPC)

Programming Model: High-performance Fortran

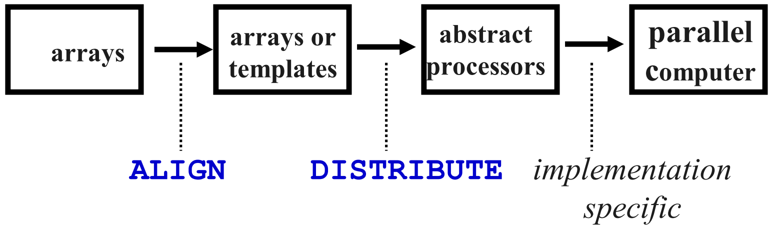
- HPF = Fortran 95 + directives
  - conceptually a single address space
    - distributed across nodes
  - source of parallelism
    - data parallelism
      - forall statements
      - rectangular arrays
      - loop-level parallelism
  - type of synchronization
    - barrier
      - statement level
      - loop level
  - all communication is generated by the compiler
    - single-side communication
    - supported in hardware

```
integer A(8), B(8), C(8)

! data-parallelism
   forall (i=1,8) do
      A(i) = B(i) + C(i)
   end do

! implicit data-parallelism
   A = B + C

! loop-level parallelism
!HPF$ INDEPENDENT
   do i = 1,8
      A(i) = B(i) + C(i)
   end do
```

HPF data distribution

- Conceptual processor grid
  - topological view of processors

```
!HPF$ processors S(4)
!HPF$ processors M(2,2)
```



- Distribution of arrays over processor grid
  - block, cyclic, or local distribution of elements
  - each dimension can be distributed independently

```
REAL A(20), B(6,8), C(6,8)
!HPF$ distribute A(BLOCK) onto M
!HPF$ distribute B(BLOCK,CYCLIC) onto P
!HPF$ distribute B(*,BLOCK) onto M
```

  - aligned to other arrays

```
!HPF$ align C(i,j) with B(i+1,j)
```

- Owner-computes rule
  - an expression that yields a result in processor *j* is computed by processor *j*

```
A(2:19) = ( A(1:18) + A(2:19) + A(3:20) ) / 3
```

HPF optimization is hard

- Simple all-pairs n-body force accumulation (n = 1000, p = 10)
  – how many communication and synchronization operations?

```
!HPF$ processors procs(10)
program hpf_pairwise_interactions

!HPF$ align Force(:,:) with Bodies(:,:)
!HPF$ align TravB(:,:) with Bodies(:,:)
!HPF$ distribute Bodies(*,BLOCK) onto procs
real Bodies(2,1000), Force(2,1000), TravB(2,1000)

    Force = 0.0
    TravB = Bodies
    do i = 1, 999
       TravB = CSHIFT(TravB,1)
       Force = Force + force_eval(Bodies, TravB)
    enddo
end
```

Whither HPF?

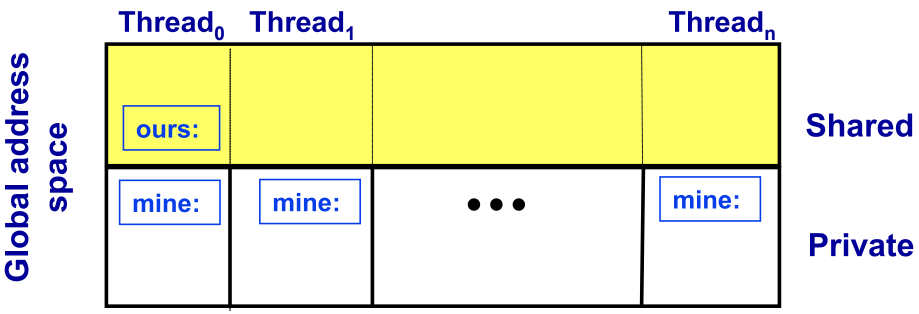
- Performance model difficult for users to understand
  - programming model (Fortran 95 semantics) quite simple
  - performance tuning requires detailed knowledge of compilation and optimization strategies

- Data distribution model too complex for compilers to optimize
  - performance requires
    - aggregation of communication
    - relaxation of barrier synchronizations
    - inferring distribution of intermediate values

- Data distribution model too restrictive
  - distribute rectangular arrays over rectangular processor grids
    - many algorithms simplified on hypercube topology
  - what about irregular applications?
    - “irregular” distribution of rectangular arrays is offered
    - but regular distribution of irregular data (e.g. trees) is what’s needed

Unified Parallel C

- UPC = C + explicit notion of locality
  - address space
    - *partitioned* global address space
      - every location in address space has *affinity* to some processor
    - a regular C pointer may reference
      - private memory
      - shared memory
        - » (dereference may have high cost)



  - source of parallelism
    - SPMD (processor centric)
  - type of synchronization
    - barriers
    - locks
    - memory consistency control – sequential or relaxed
  - most communication is implicit
    - distribution of shared arrays is much simpler than HPF
      - conceptuallly 1-D array of processors, with cyclic, block-cyclic, or block distribution
    - message passing / one sided communication generated by UPC compiler

UPC extensions to C

- Processor count and processor id
  - compile-time symbolic values
    - `THREADS` - number of processors
    - `MYTHREAD` - thread id (0 ≤ MYTHREAD < THREADS)
  - compilation environment
    - static – number of processors fixed at compile time (not really used)
    - dynamic – number of processors supplied at run time (always used)

- `shared` qualifier for type declarations
  - elements of a shared array distributed across processors

- forall construct

```
upc_forall (i = 0; i < N; i++; <affinity>) {…}
```

UPC declarations

- Shared array declarations
  - `shared [`*blocksize*`]` *<decl>* `[`*count*`]`
    - blocksize defaults to 1
  - specifies block cyclic distribution of *<decl>* in shared memory

- Examples

  `shared int a`
  - Single shared memory location (with affinity to thread 0)

  `int b`
  - private memory location at each thread

  `shared int x[THREADS]`
  - One element per thread

  `shared [3] int x[N]`
  - N/p elements per thread, cyclic(3) dist

  `shared int y[10][THREADS]`
  - single array of 10 elements per thread (block distribution)

UPC Hello world

- Any legal C program is also a legal UPC program
  - When run as a UPC program with p threads, it will run p copies of the program

```
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

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

Simple UPC example

- Vector addition using upc_forall

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

Effect of Array Distributions in UPC

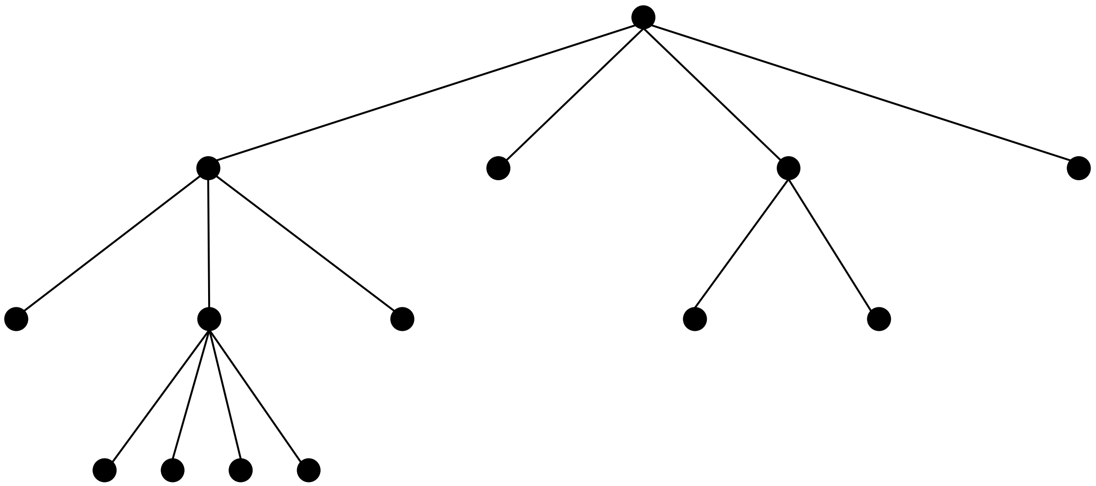
- The cyclic distribution of an array is typically stored in one of two ways
  - Distributed memory: each processor has a chunk of memory
    - Thread 0 would have elements: 0,THREADS, THREADS*2,… in a chunk
  - Shared memory: array elements appear consecutively in memory
    - Thread 0 would reference successive elements with stride THREADS
      - What performance problem is there with the latter?
      - What if this code was instead doing nearest neighbor averaging?

- Vector addition example can be rewritten using block distribution

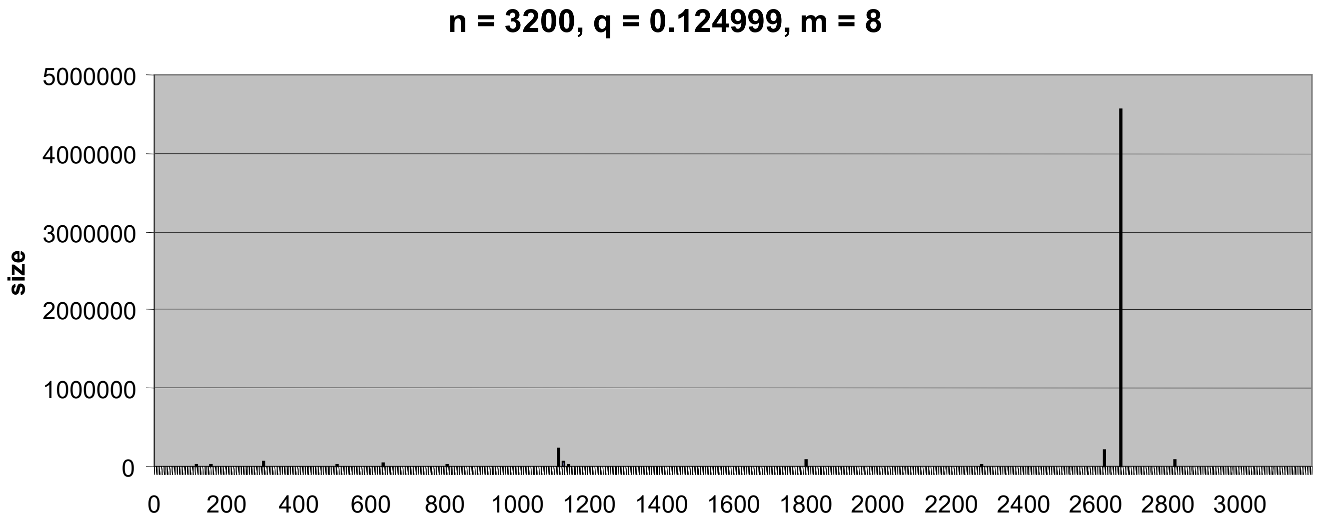
```
#define N 100*THREADS
shared int [*] v1[N], v2[N], sum[N];  // blocked distribution
void main() {
   int i;
   upc_forall(i=0; i<N; i++; &v1[i])
         sum[i]=v1[i]+v2[i];
   }
```

Example: Unbalanced Tree Search (UTS)

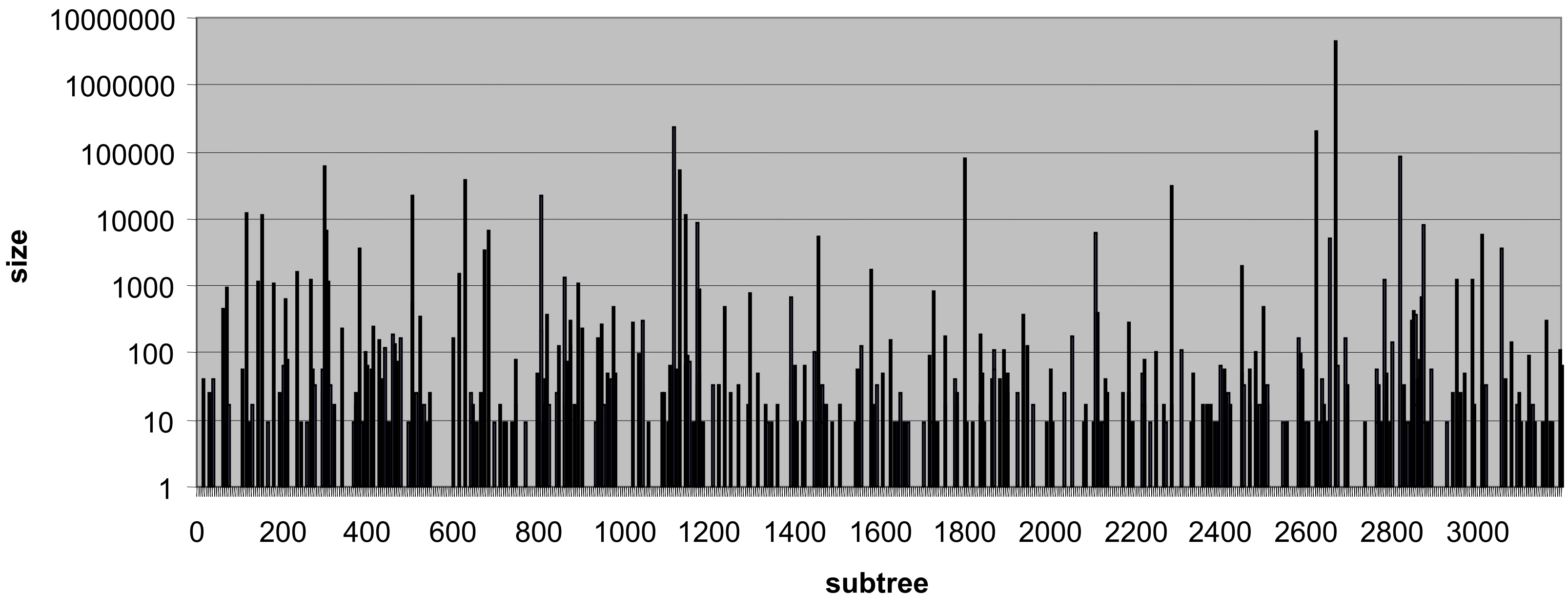
- Problem description
  - count number of nodes in a tree
    - tree is implicitly defined
  - parallel depth-first search implementation
    - traverse subtrees in parallel counting size and combine on completion
  - unbalanced trees
    - subtrees have large variation in size



Unbalanced tree search



distribution of 3200 (q,m) tree sizes



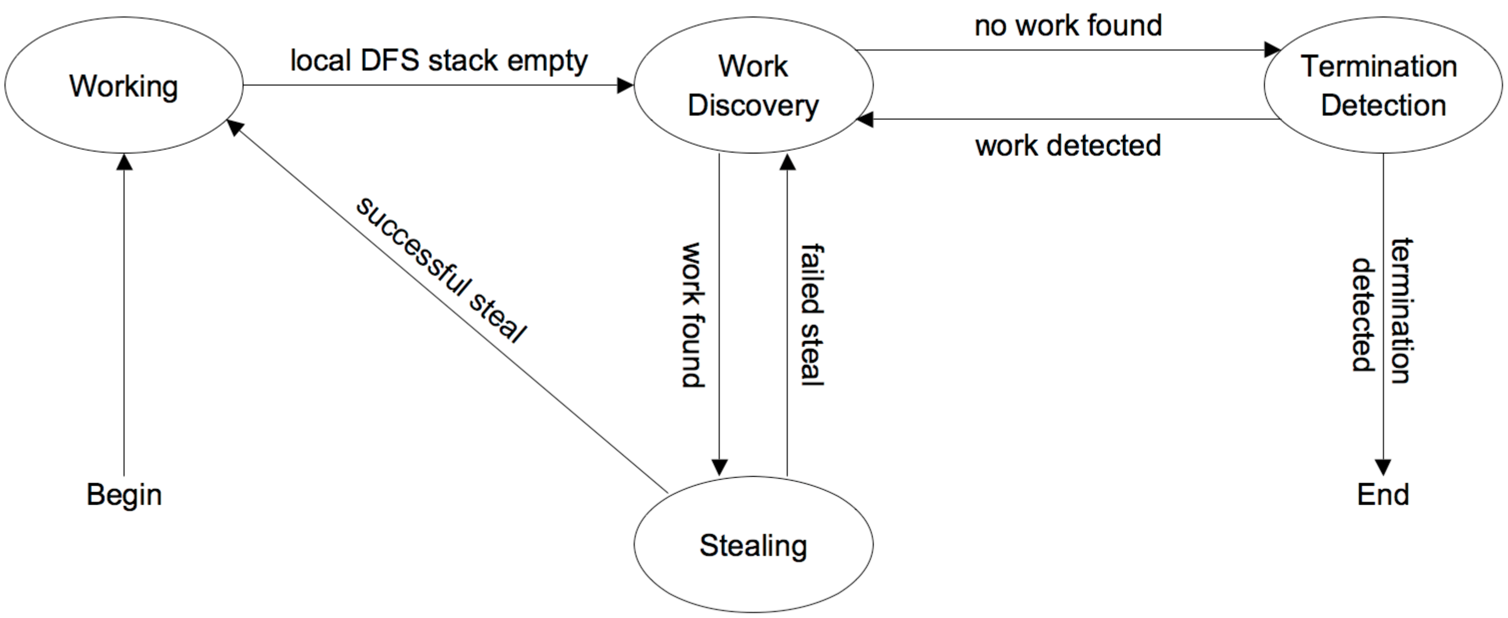
Search strategy

- P processors explores a (q,m) tree
  - starting configuration
    - proc 0 has root node descriptor
    - other procs have no tree nodes

  - tree is implicitly generated
    - Binomial tree (q,m)
      - if $qm < 1$ generates a finite tree with expected size $\frac{1}{1-qm}$
      - each node has a 20 byte descriptor
    - given a tree node t, generate m children with probability q
      - use node descriptor as seed in random number generator
      - children descriptors are determined using SHA-1 hash of parent
    - perform depth-first search of each child
      - uses a stack
    - when the stack is empty
      - steal work from another processor's stack
      - ideal for Cilk execution model

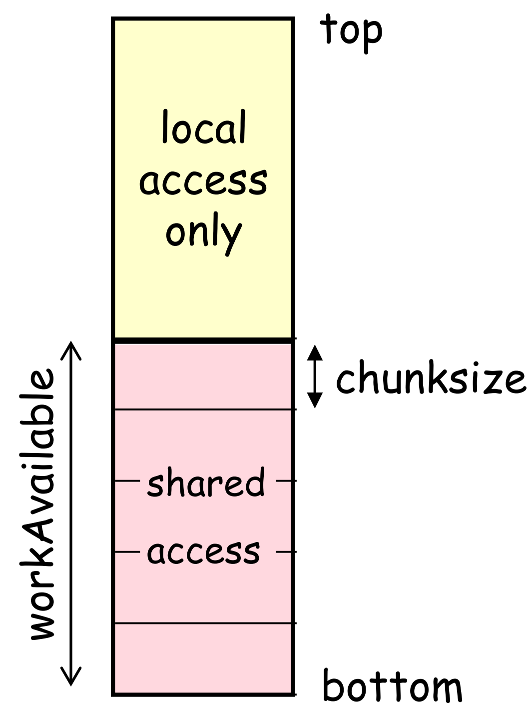
UTS: basic operation

- Basic operation of a thread is shown in the state diagram below:



StealStack

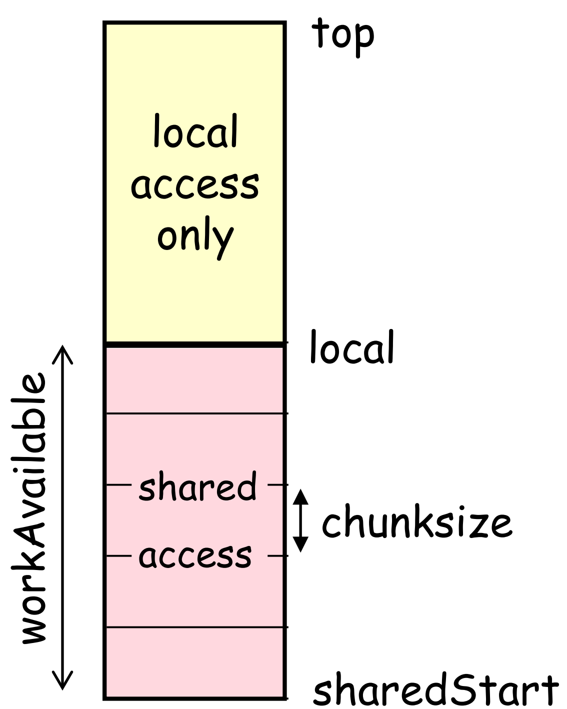
- Efficient shared and local access to a stack
  - stack of nodes
    - local access only at top of stack
    - shared area at bottom of stack
  - shared area
    - protected by lock in thread *i*
    - manipulated in chunks
      - thread *i* release a chunk from local portion into shared portion
      - thread *i* acquire a chunk from shared portion into local portion
      - thread *j* steals a chunk from bottom of shared portion of stack in thread *i*
  - shared variable workAvailable
    - current size of shared portion



UPC implementation

Representation of shared stack

```
/* StealStack data type */
struct stealStack_t
{
  int workAvail;
  int sharedStart;
  int local;
  int top;
  upc_lock_t *stackLock;
  Node stack[MAXSTACKDEPTH];
};
typedef struct stealStack_t StealStack;

/* StealStack for each thread */
shared StealStack stealStack[THREADS];

/* direct access to stack with affinity */
myStack = (StealStack *) stealStack[MYTHREAD];
```



UPC implementation

Local push/pop

- no locking
- shared stack accessed through local pointer
  - no UPC overhead

```
/* local push */
void push(StealStack *s, Node *c) {
  if (s->top >= MAXSTACKDEPTH)
    error("StealStack::push  overflow");
  memcpy(&s->stack[s->top], c, sizeof(Node));
  s->top++;
  s->maxDepth = max(s->top, s->maxDepth);
}
```

UPC implementation

Steal from thread i

```
/* steal k values from thread i onto this thread's stack
 * return false if k vals are not avail in thread i
 */
int steal(StealStack *s, int i, int k) {
  int victimLocal, victimShared, victimWorkAvail, ok;

  /* lock stack in thread i and try to reserve k elts */
  upc_lock(stealStack[i].stackLock);
  victimLocal = stealStack[i].local;
  victimShared = stealStack[i].sharedStart;
  victimWorkAvail = stealStack[i].workAvail;
  ok = victimWorkAvail >= k;
  if (ok) {
    /* reserve k values */
    stealStack[i].sharedStart =  victimShared + k;
    stealStack[i].workAvail = victimWorkAvail - k;
  }
  upc_unlock(stealStack[i].stackLock);
```

UPC implementation

Steal from stack *i* (contd.)

- data movement does not hold lock

```
/* if k elts reserved, move them to local portion of this stack */
  if (ok) {
    upc_memcpy(&stealStack[MYTHREAD].stack[s->top],
               &stealStack[i].stack[victimShared],
               k * sizeof(Node)
               );
    s->top += k;
    s->nSteal++;
  }
  else
    s->nFail++;
  return (ok);
}
```

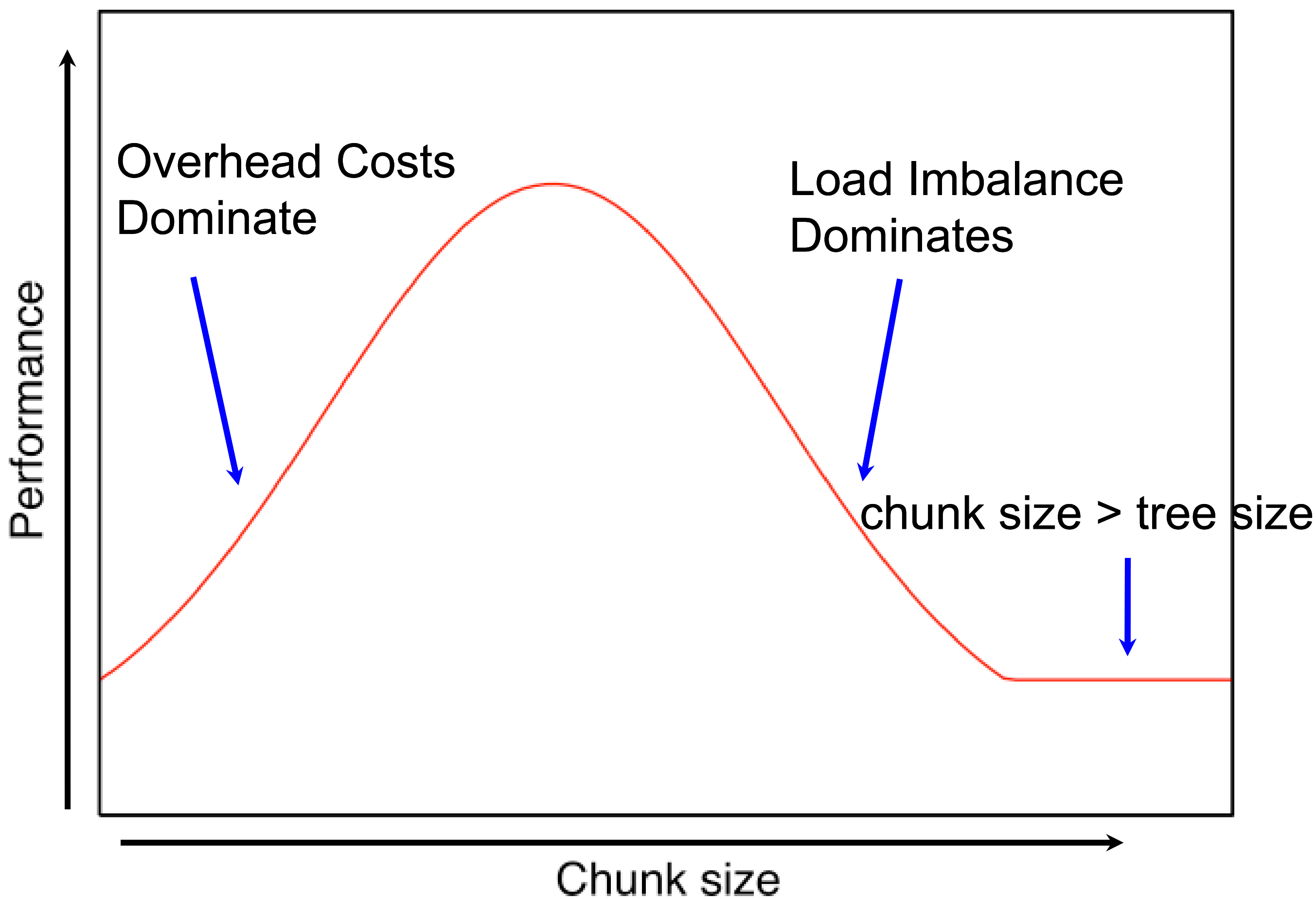
What is the optimal choice for chunksize?

- Small chunks
  - may not yield much work
    - hence may not amortize time to move
  - have higher manipulation overheads
    - locking and unlocking

- Large chunks
  - are available less frequently
    - hence may not balance load
  - Depth first stack length l satisfies

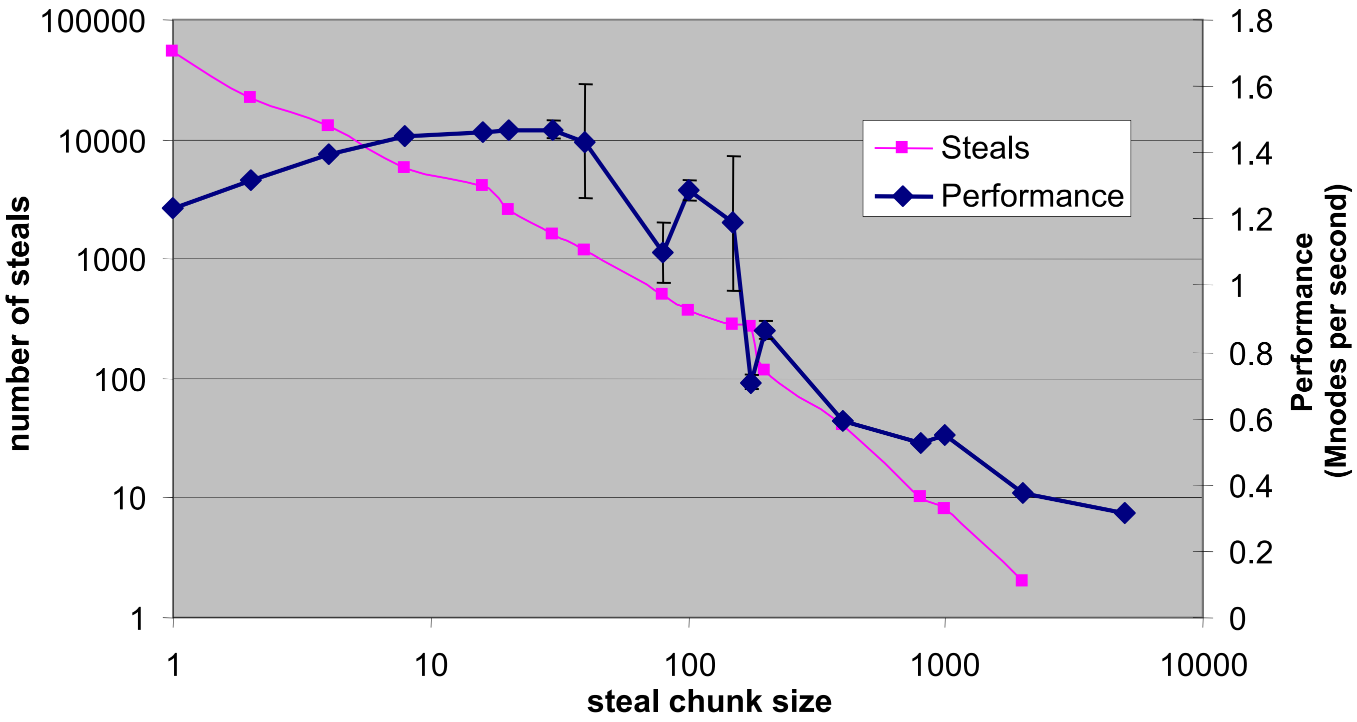
$$\Pr(l \geq T) < \left(\frac{E(n)^2}{1 - E(n)^2} \right) \cdot \frac{1}{T}$$

UTS: Granularity of Work Stealing



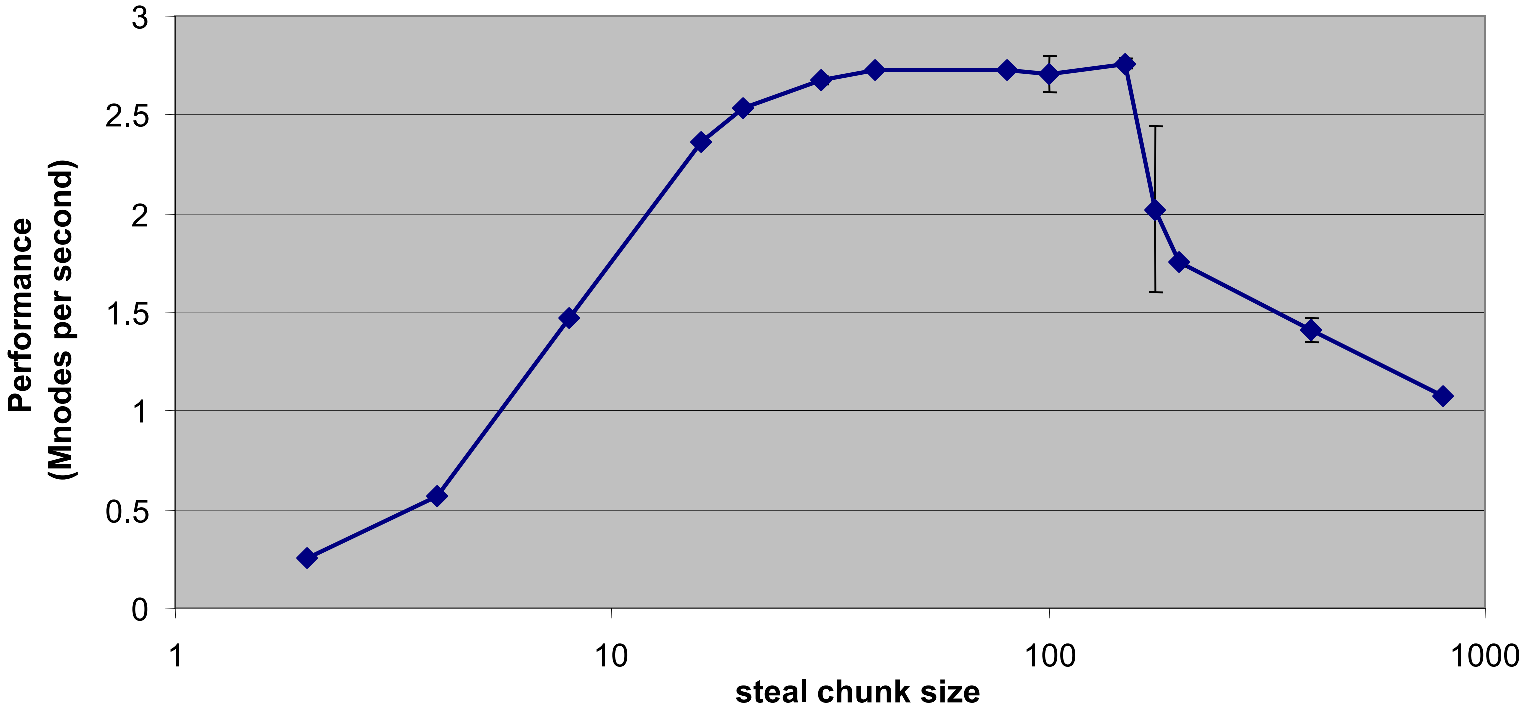
Chunk size vs. performance

- Intrepid UPC compiler on Origin 2000
  - 8 threads
  - single group, 32×100 trees, 9.5M nodes total



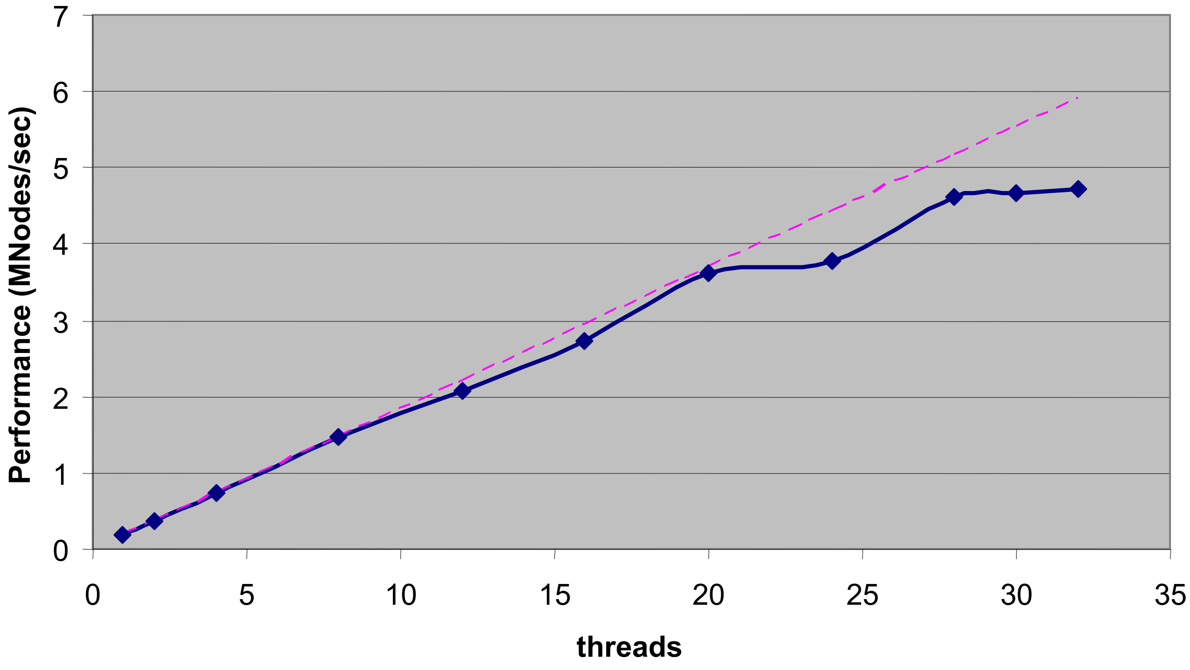
Chunk size dependence on communication costs

- Compaq UPC compiler V1.7 on ORNL AlphaServer SC
  - 8 threads
  - same tree and parameters



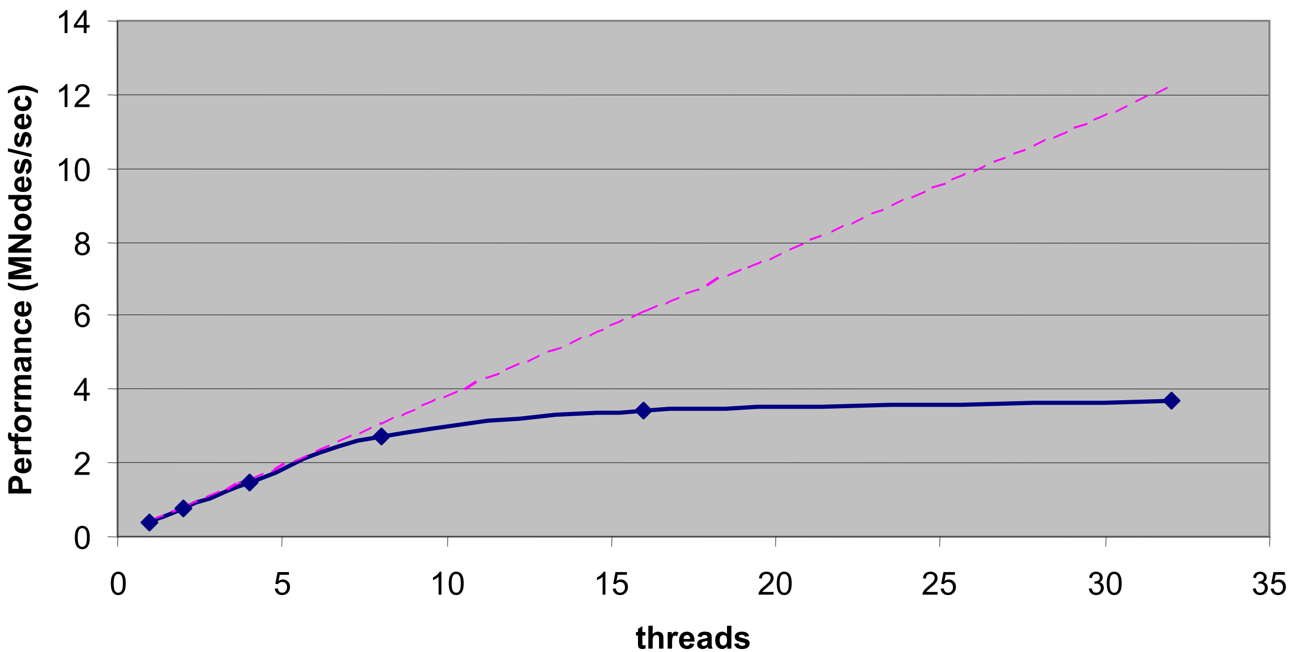
Shared-memory implementation scaling

- Origin 2000
  - 1 - 32 processors
  - chunksize = 20



Distributed memory implementation scaling

- ORNL AlphaServer SC Distributed memory
  - 1,2,4,8,16,32 processors
  - chunksize = 100
- Remote locking on distributed memory is expensive
  - Even though lock is local to “victim thread,” victim is delayed during slow remote accesses by other threads



Scalable Distributed Memory UPC Implementation

- Request and response protocol uses asynchronous remote reads & writes to shared variables
  - Response returns a pointer to the work reservation (if work available)
  - Working threads never wait on locks

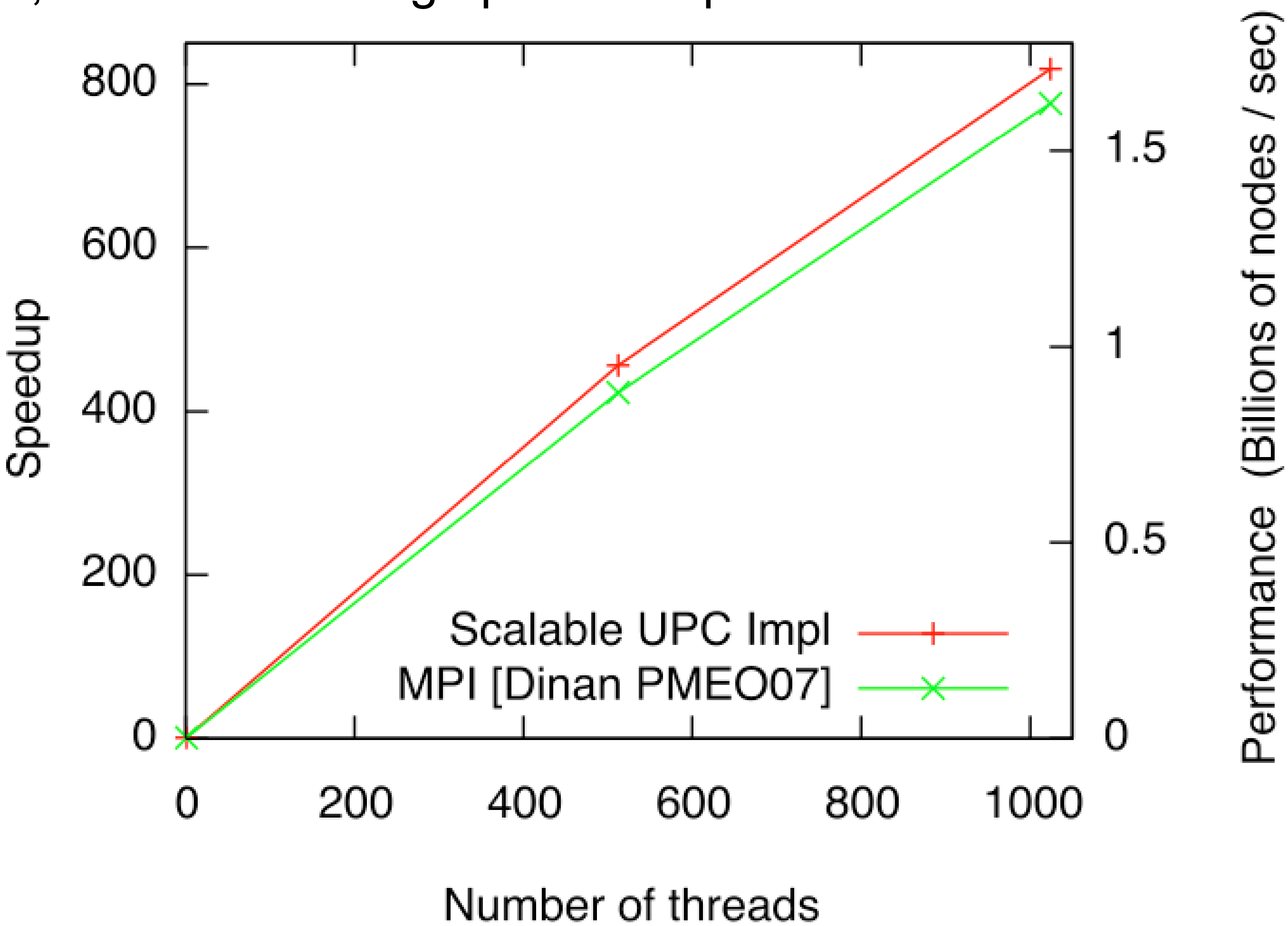
- One-sided communication to transfer work

Scalable UPC Implementation: Stealing Protocol

Protocol Phase	Victim (Working Thread) Memory	Thief (Idle Thread) Memory
Thief probes for work	Thief reads WK_AVAILABLE	
Thief attempts request (test-and-set)	Thief sets REQ_LOCK	
	Thief reads REQ_ID	
	Thief writes REQ_ID	
	Thief releases REQ_LOCK	
Victim detects request (poll)	Victim reads REQ_ID	
	Victim resets REQ_ID	
Victim reserves work for thief		Victim writes WK_PTR
Thief detects response (spin)		Thief reads WK_PTR
Thief transfers work	Thief reads nodes at WK_PTR	
		Thief resets WK_PTR

Scaling

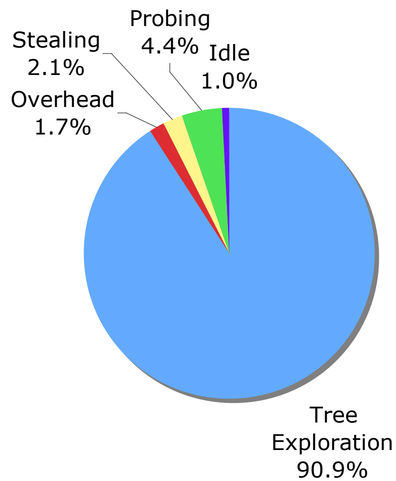
- 157 billion node tree, 1024 processors
- 85,000 work stealing operations per second



Where Does the Time Go?

Cluster 1 (2.66Ghz Xeon)

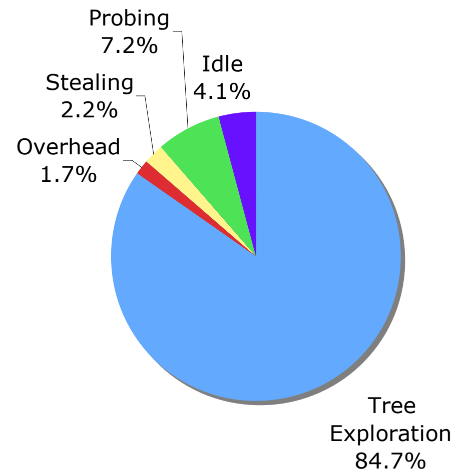
256 Threads
10.6B Node Tree



604K steals
115 steals / thread / sec
94.5% of steal attempts succeed
Node evaluation time: 0.393 μs

Cluster 2 (2.4 Ghz Xeon)

1024 Threads
157B Node Tree



8.14M steals
86 steals / thread / sec
94% of steal attempts succeed
Node evaluation time: 0.459 μs

Summary: high-level PPLs for distributed memory

- Emerging model
  - Partitioned global address space model
    - Explicit notion of locality
      - Control over data distribution
    - One-sided communication
  - Current examples
    - Global Arrays (C Library)
    - UPC (C)
    - Co-Array Fortran (Fortran)
    - Titanium (Java)
  - Future “High Productivity” parallel programming languages
    - X10 (Java + tasks + PGAS)