Lecture 11

Parallel Programming Languages

Announcements

- A3 due Tuesday
- A3 turnin enabled

Today's lecture

- Parallel Programming Languages
  - UPC
  - Cilk

Unified Parallel C (UPC)

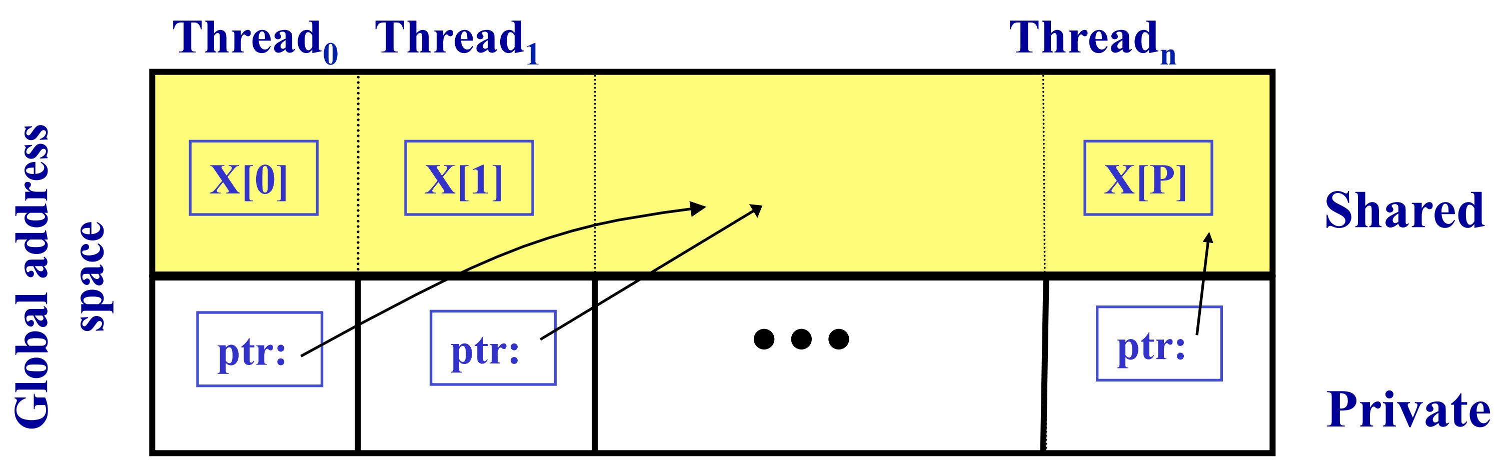
The Berkeley UPC Group: C. Bell, D. Bonachea, W. Chen, J. Duell,
P. Hargrove, P. Husbands, C. Iancu, R. Nishtala, M. Welcome, K. Yelick

http://upc.lbl.gov

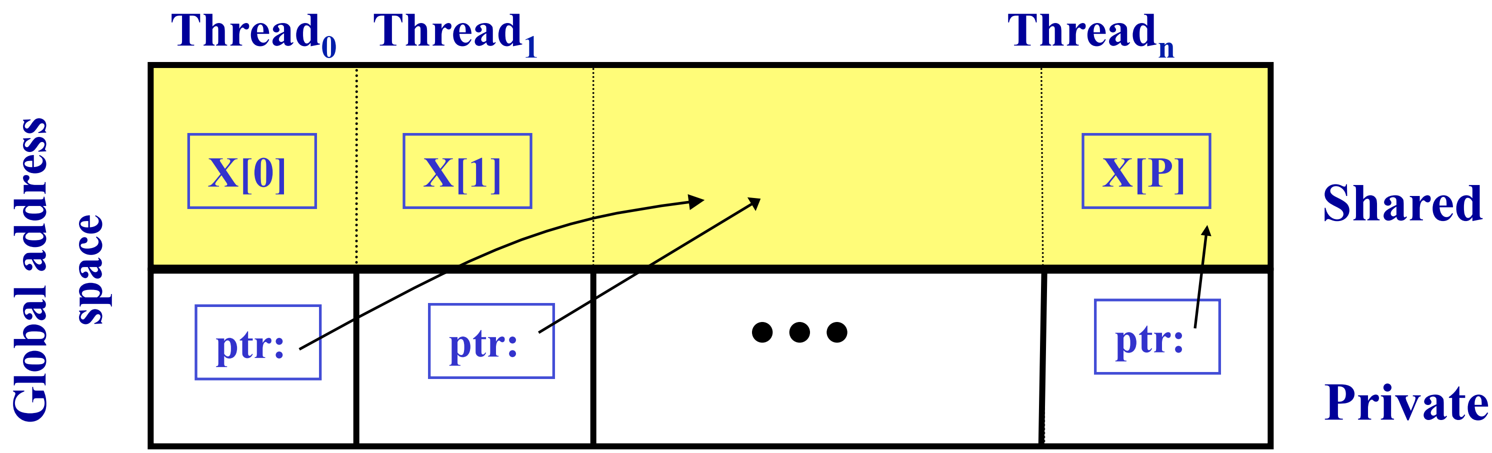
Slides edited by K. Yelick, T. El-Ghazawi, P. Husbands, C.Iancu

UPC

- An explicit parallel extension of ANSI C
- SPMD parallelism based on threads
- PGAS Language: Partitioned Global Address Space
  - Address space is logically partitioned: local & remote
  - Others: Co-Array Fortran, Titanium, Chapel, Fortress, X10
- Programmer control over performance critical decisions: data layout and data motion

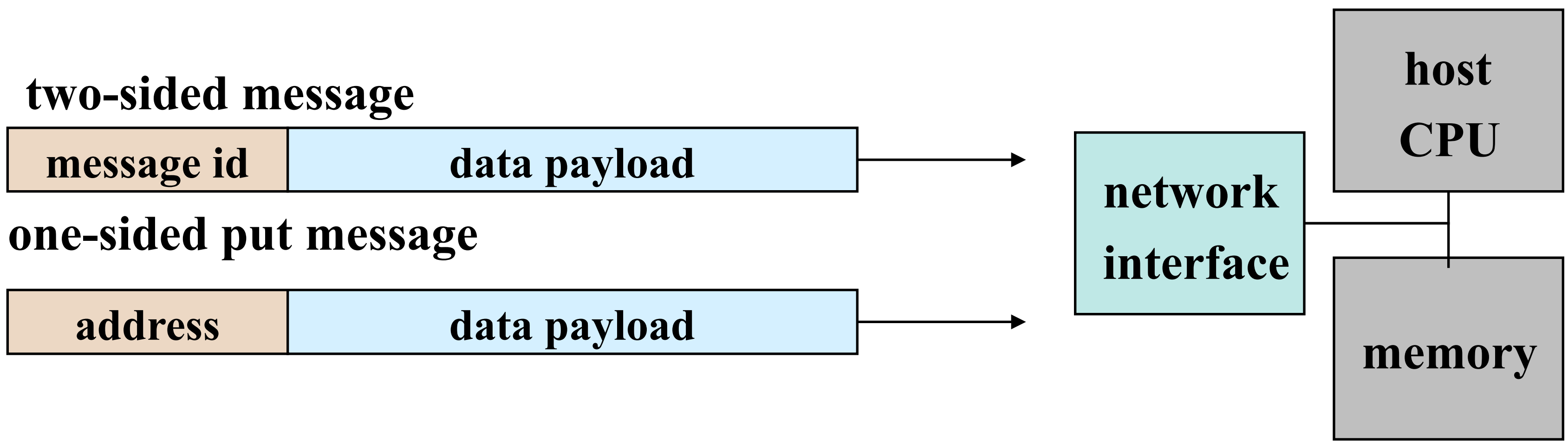


Partitioned Global Address Space



- Shared memory logically partitioned over processors
- Remote memory directly accessible, without hardware caching
- *One-sided communication:* put() or get() to remote memory
- Some models have a separate private memory area

One-Sided vs Two-Sided Communication



- A two-sided message needs to be matched at the recipient to identify memory the address to put the data and in some cases ensure space at receiving end
  - Offloaded to Network Interface
- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU

UPC Execution Model

- Thread model
  - # threads specified at compile- or run-time
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
  - **`upc_barrier`** is a global synchronization
  - **`upc_forall`** parallel loop construct
- Two compilation modes
  - Static threads mode:
    - # THREADS is specified at compile time by the user
    - The program may use THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

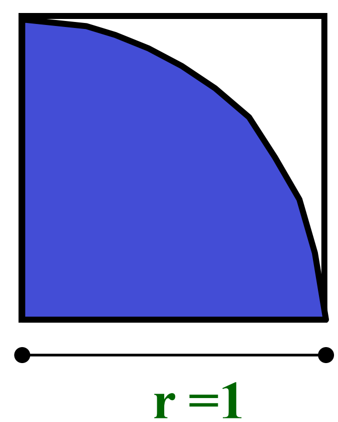
Hello World in UPC

- Any legal C program is also legal UPC
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Parallel hello world:

```
#include <upc.h>
#include <stdio.h>
main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then inside the circle
- Compute ratio
  - # points inside / # points total
  - $\pi = 4*\text{ratio}$



Pi in UPC

- Independent estimates of pi:

```
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;

    if (argc != 2)trials = 1000000;
    else trials = atoi(argv[1]);

    srand(MYTHREAD*17);

    for(i=0; i < trials; i++)hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```

Each thread gets a private copy

Each thread sees the input arguments

Initialize random number geneartor

Each thread calls “hit” separately

Helper Code

- Throw dart and compute where it hits

```
int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
       return(1);
  } else {
       return(0);
  }
}
```

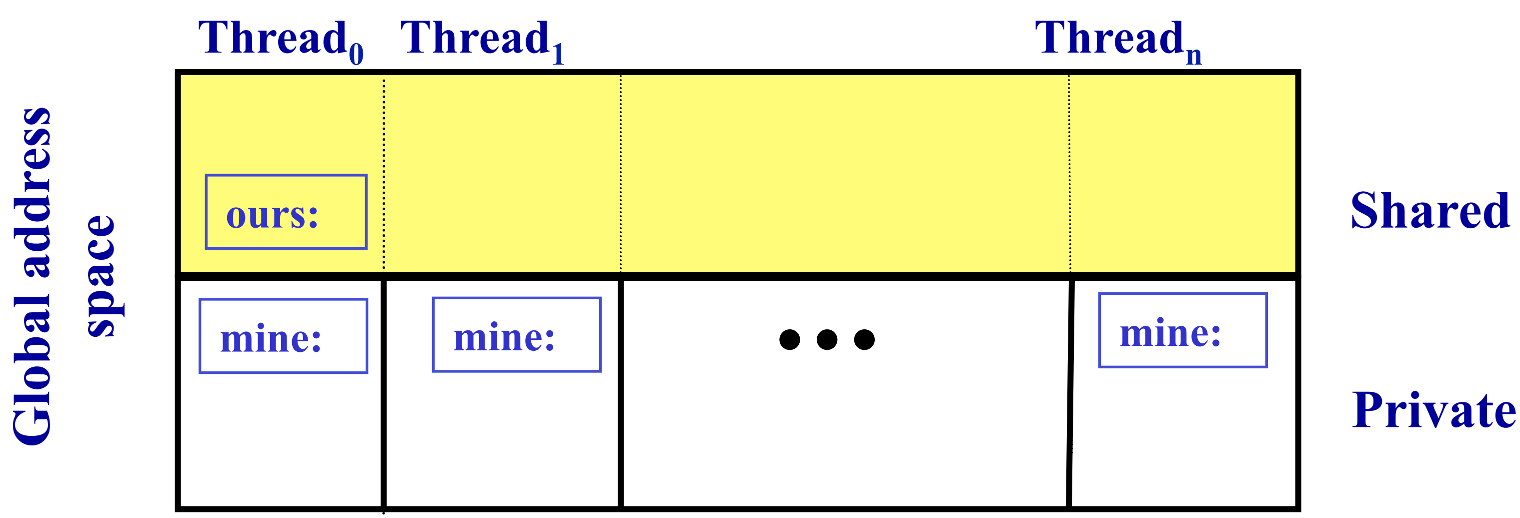
Shared vs. Private Variables

Private vs. Shared Variables

- Normal C variables and objects are allocated in the private memory space for each thread
- Shared variables are allocated only once, by thread 0

```
shared int ours;  // use sparingly
int mine;
```

- Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static. Why?



Using shared memory

Where is the bug?

shared variable to record hits

divide work up evenly

accumulate hits

```
shared int hits;
main(int argc, char **argv) {
  int i, my_trials = 0;
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1)/THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    hits += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
   printf("PI estimated to %f.",4.0*hits/trials);
    }
}
```

UPC Synchronization

Fixing the bug in Pi

```
shared int hits;
  main(int argc, char **argv) {
      int i, my_hits, my_trials = 0;
      upc_lock_t *hit_lock = upc_all_lock_alloc();
      int trials = atoi(argv[1]);
      my_trials = (trials + THREADS - 1)/THREADS;
      srand(MYTHREAD*17);
      for (i=0; i < my_trials; i++)
         my_hits += hit();
      upc_lock(hit_lock);
      hits += my_hits;
      upc_unlock(hit_lock);
      upc_barrier;
      if (MYTHREAD == 0)
        printf("PI: %f", 4.0*hits/trials);
   }
```

create a lock

accumulate hits locally

accumulate across threads
within a critical section

Shared Array Version

- Alternative fix to the race condition
- Each thread updates a separate counter
  - But residingin a shared array
  - One thread computes the global sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
  … declarations an initialization code omitted
  for (i=0; i < my_trials; i++)
    all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++) hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
}
```

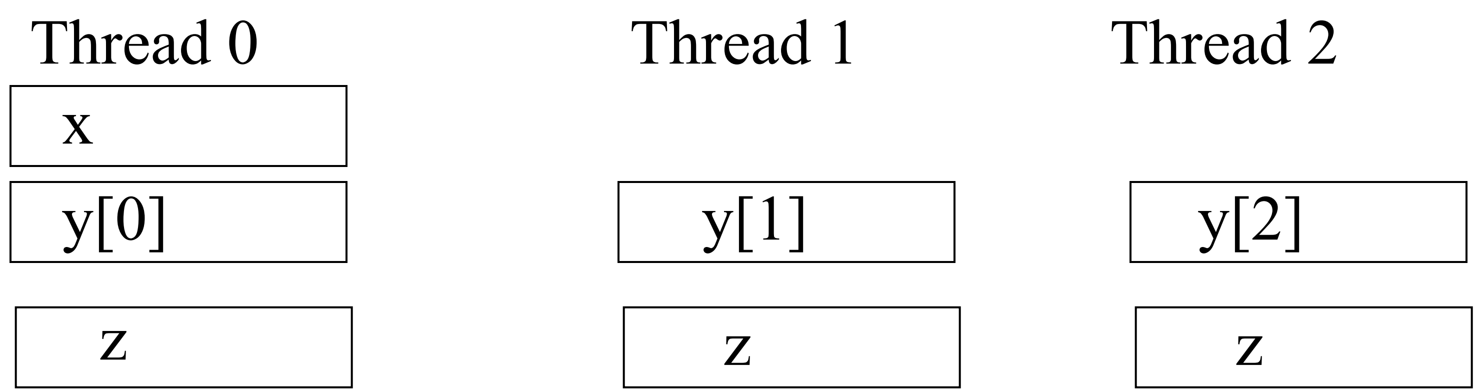
all_hits is shared by all processors, just as hits was

update element with local affinity

Shared arrays

- Shared arrays are spread across the threads
- Assume THREADS = 3

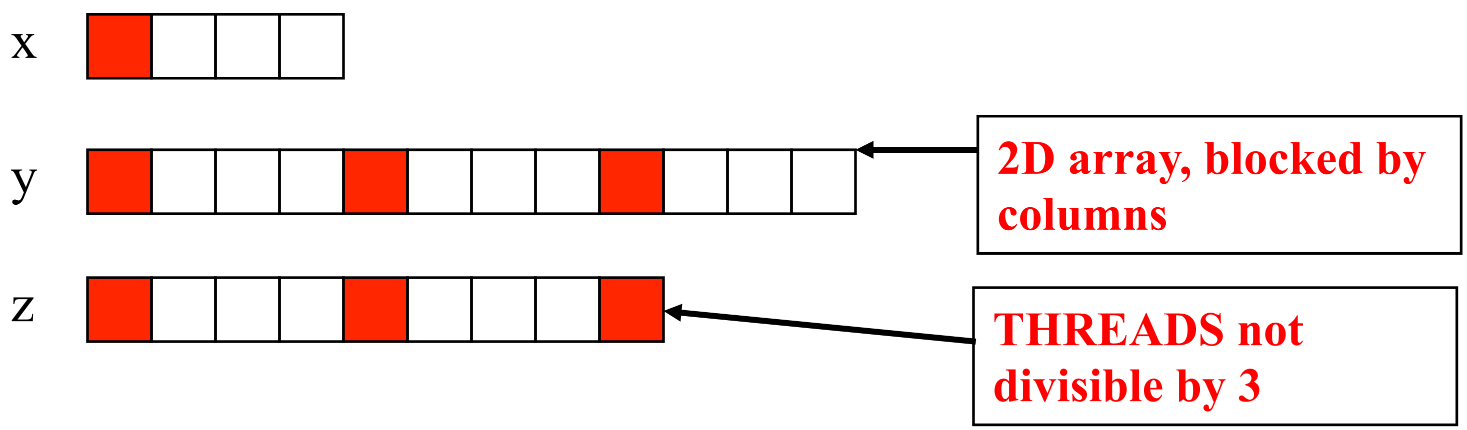
```
shared int x;            /*x has affinity to thread 0 */
shared int y[THREADS];
int z;
```



Shared Arrays are Cyclic by Default

```
const int THREADS=4;
shared int x[THREADS]       /* 1 element per thread */
shared int y[3][THREADS]    /* 3 elements per thread */
shared int z[3][3]          /* 2 or 3 elements per thread */
```

- Red elements have affinity to thread 0



Using collectives

- **The previous version of Pi is not scalable**
  - **On a large # of threads, the locked region will be a bottleneck**
- **Use a reduction for better scalability**

```
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ...
    for (i=0; i < my_trials; i++)
       my_hits += hit();
    my_hits =           // type, input, thread, op
       bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*my_hits/trials);
  }
```

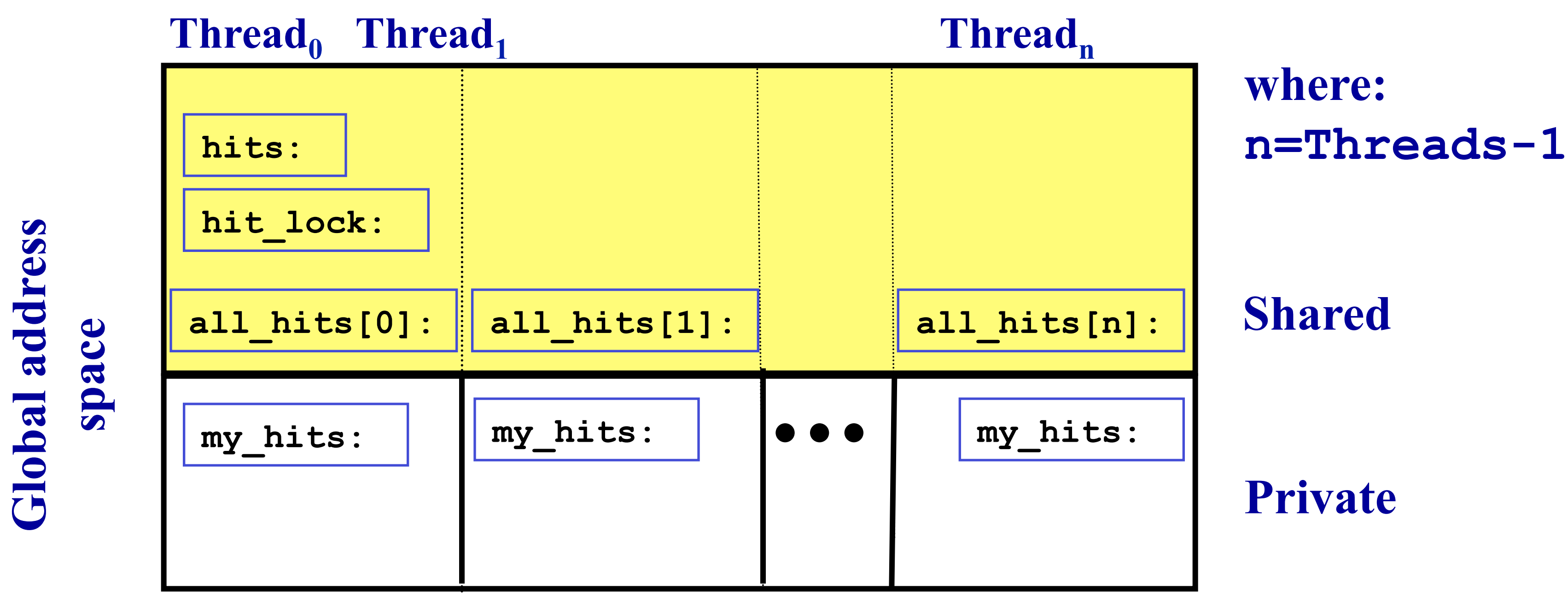
Berkeley collectives

no shared variables

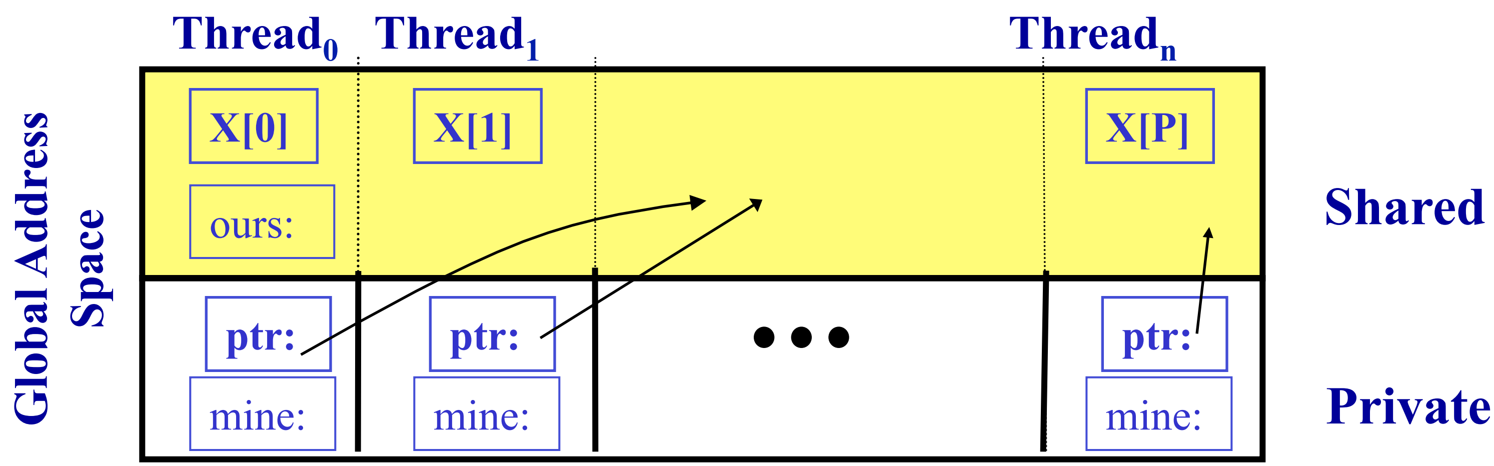
barrier implied by collective

Private vs. Shared Variables

- Private scalars (`my_hits`)
- Shared scalars (`hits`)
- Shared arrays (`all_hits`)
- Shared locks (`hit_lock`)



Data Distribution



- Distinguish memory spaces via extensions to the type system (**`shared`** qualifier)

```
shared int ours;
shared int X[THREADS];
shared int *ptr;
int mine;
```

- Data in shared address space:
  - **Static**: scalars (T0), distributed arrays
  - **Dynamic**: dynamic memory management

  (**`upc_alloc, upc_global_alloc, upc_all_alloc`**)

UPC Pointer Implementation

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)

Virtual Address	Thread	Phase

- Example: Cray T3E implementation

Phase	Thread	Virtual Address
63 49	48 38	37 0

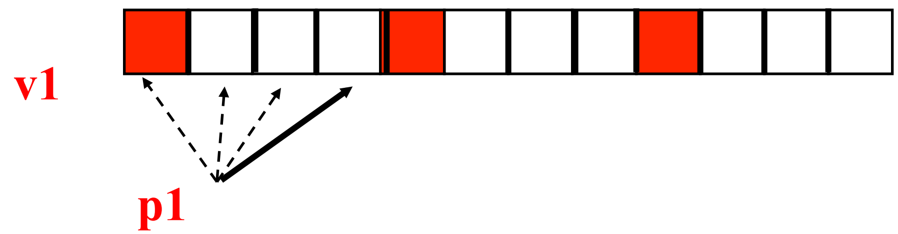
- Pointer arithmetic can be expensive in UPC

Arrays vs. Pointers to Shared

- In the C tradition, arrays can be accessed through pointers
- Vector addition using pointers

```
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
   int i;
   shared int *p1, *p2;

   p1=v1; p2=v2;
   for (i=MYTHREAD; i<N;
         i+=THREADS, p1+=THREADS, p2+=THREADS )
        sum[i]= *p1 + *p2;
   }
```



UPC Pointers

Where does the pointer point?

Where does the pointer reside?

	Local	Shared
Private	PP (**p1**)	PS (**p3**)
Shared	SP (**p2**)	SS (**p4**)

```
int *p1;              /* private pointer to local memory */
shared int *p2;       /* private pointer to shared space */
int *shared p3;       /* shared pointer to local memory */
shared int *shared p4; /*shared pointer to shared space */
```

Common Uses for UPC Pointer Types

`int *p1`

- Fast, just like C pointers
- Use to access local data in part of code performing local work
- Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

`shared int *p2: private ptr to shared space`

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication

`int *shared p3:`

- **`shared ptr to local memory`**
- Not recommended

`shared int *shared p4;`

- **`shared pointer to shared space`**
- Use to build shared linked structures, e.g., a linked list

UPC Pointer Usage Rules

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not the other way around!
- When casting a pointer-to-shared to a pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast

Work Distribution

Example: Vector Addition

```
#include <upc_relaxed.h>
#define N 100*THREADS
```

cyclic layout

```
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
                        sum[i]=v1[i]
+v2[i];
}
```

owner computes

```
upc_forall(i=0; i<N; i++; &v1[i])
    sum[i]=v1[i]+v2[i];
```

Work Distribution: upc_forall()

- UPC adds a special type of loop

```
upc_forall(init; test; step; affinity)
    statement;
```

- **Owner computes rule:** loop over all, work on those owned by you

```
upc_forall(i=0; i<N; i++; &v1[i])
     sum[i]=v1[i]+v2[i];
```

- Declares that iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression `establishes` which iterations to run on each thread
  - Integer: `affinity%THREADS == MYTHREAD`
  - Pointer: `upc_threadof(affinity) == MYTHREAD`
- Syntactic sugar for:

```
for(i=MYTHREAD; i<N; i+=THREADS)
    …
 for(i=0; i<N; i++)
  if (MYTHREAD == i%THREADS)
     …
```

Distributed Arrays

Data Layout

- Data layout controlled via type system extensions (layout specifiers)
  - [0] or [] (indefinite layout, all on 1 thread):

    ```
    shared [] int *p;
    ```
  - Empty (cyclic layout) :

    ```
    shared int array[THREADS*M];
    ```
  - [*] (blocked layout):

    ```
    shared [*] int array[THREADS*M];
    ```
  - [b] or [b1][b2]…[bn] = [b1*b2*…bn] (block cyclic)

    ```
    shared [b] int array[THREADS*M];
    ```
- Element `array[i]` has affinity with thread (non-arrays to thread 0)

  ```
  (i / b) % THREADS
  ```
- Layout determines pointer arithmetic rules
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping
- Introspection (`upc_threadof, upc_phaseof,upc_blocksize`)

Blocked Layouts

```
#define N 100*THREADS
  shared int [*] v1[N], v2[N], sum[N];

  void main() {
      int i;
      upc_forall(i=0; i<N; i++; &v1[i])
                  sum[i]=v1[i]+v2[i];
  }
```

blocked layout

Blocking of Shared Arrays

- We can add a block size to the declaration (block cyclic)

```
shared [block-size] type array[N];

    shared [4] int a[16];
```

Blocking Shared Arrays

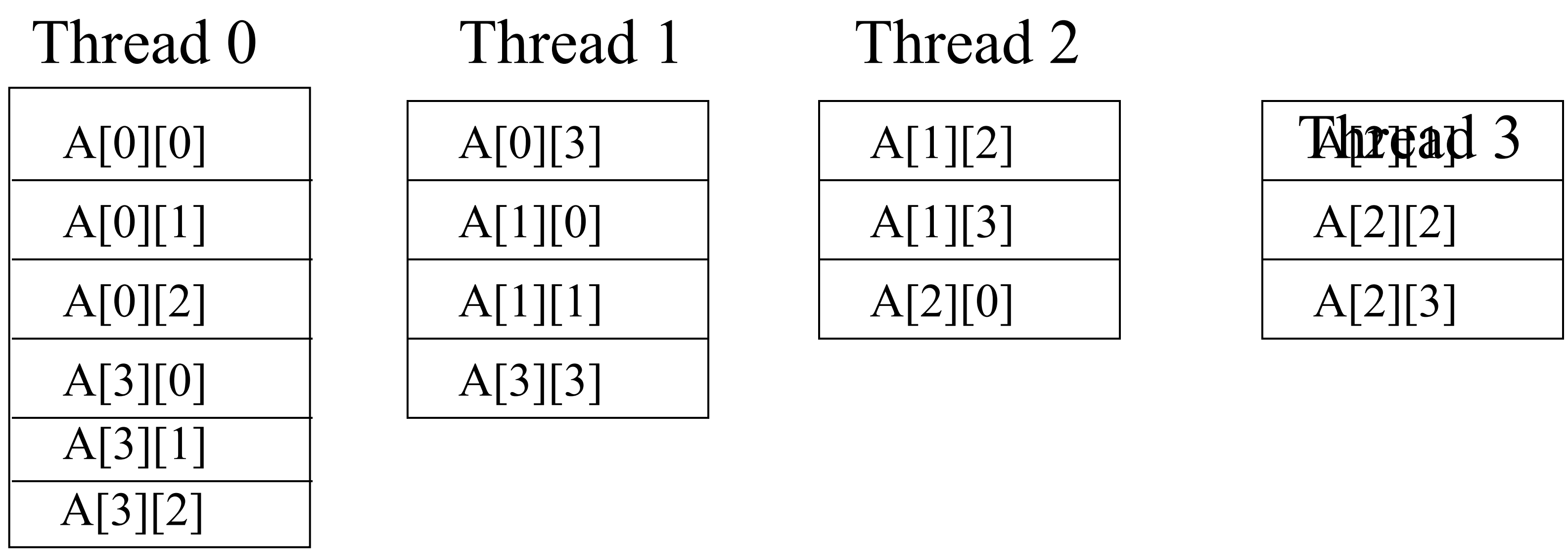
- Block size and THREADS determine affinity
- The term affinity means in which thread's local shared-memory space, a shared data item will reside
- Element *i* of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \bmod THREADS$$

Shared and Private Data

Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```



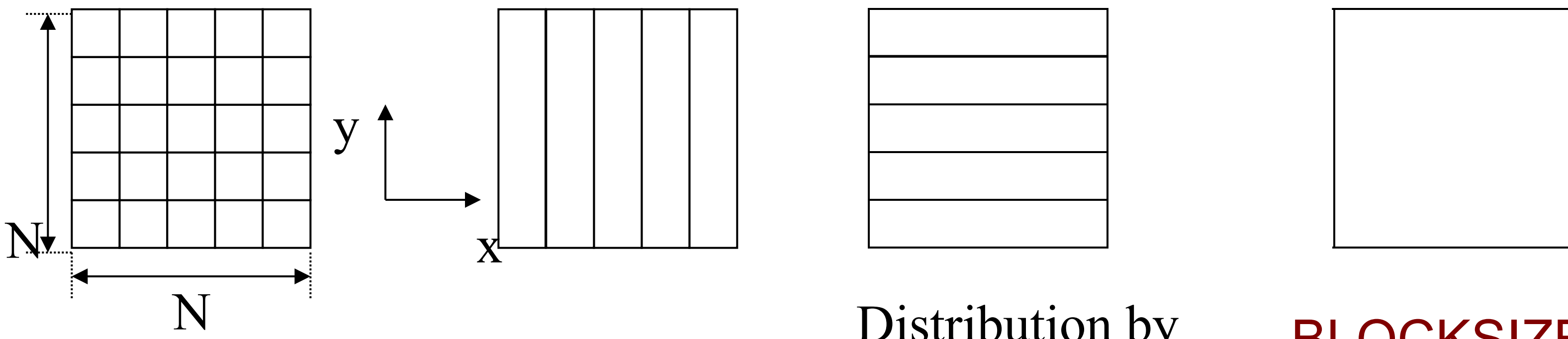
Blocked layouts

```
shared int A[4][THREADS];
```

Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

Distributing Multidimensional Data

shared [BLOCKSIZE] double G[N][N];



Default

BLOCKSIZE=1

Column Blocks

N/THREADS

Distribution by Row Block

N

BLOCKSIZE=N*N
or
BLOCKSIZE = ∞

- Contiguous memory layout of C multidimensional arrays
- Distribution depends on the value of BLOCKSIZE

2D Array Layouts in UPC

row `shared [m] int a1 [n][m];`

Block row `shared [k*m] int a2 [n][m];`

- If `(k + m) % THREADS = 0` them a3 has a row layout

  `shared int a3 [n][m+k];`

- To get more general 2D blocked layouts, we need to add dimensions.
- Assume `r*c == THREADS`

  `shared [b1][b2] int a5 [m][n][r][c][b1][b2]`

- or equivalently

  `shared [b1*b2] int a5 [m][n][r][c][b1][b2]`

Matrix Multiplication in UPC

- Given two integer matrices A(NxP) and B(PxM), we want to compute C =A x B.
- Entries c_{ij} in C are computed by the formula:

$$c_{ij} = \sum_{l=1}^{p} a_{il} \times b_{lj}$$

Serial C code

```
int a[N][P], c[N][M], b[P][M]};

void main (void) {
     int i, j , l;
     for (i = 0 ; i<N ; i++) {
       for (j=0 ; j<M ;j++) {
         c[i][j] = 0;
         for (l = 0 ; l<P ; l++)
           c[i][j] += a[i][l]*b[l][j];
       }
     }
}
```

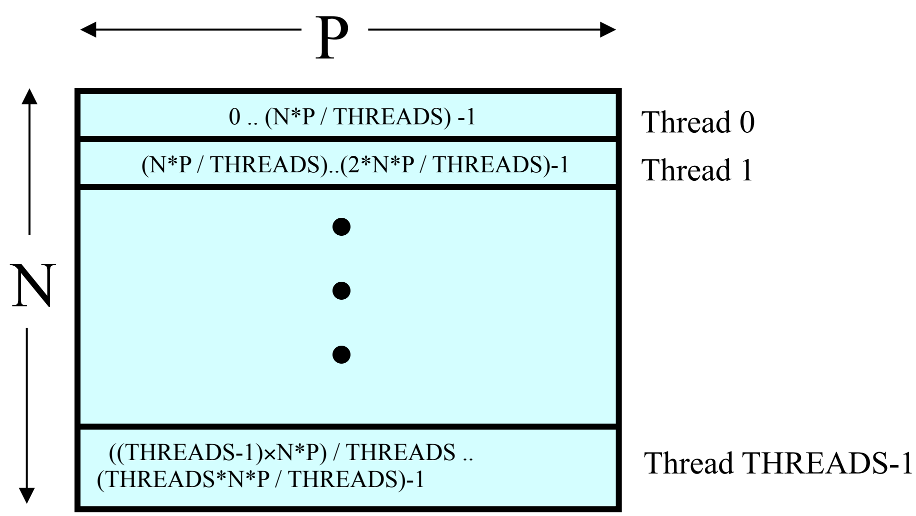
UPC Matrix Multiplication Code

```
shared [N*P/THREADS] int a[N][P] = {… }, c[N][M];
// data distribution: a and c are blocked shared
shared [M/THREADS] int b[P][M] = {…};
//column-wise blocking
void main (void) {
     int i, j , l; // private variables
     upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
      //work distribution
      for (j=0 ; j<M ;j++) {
          c[i][j] = 0;
          for (l= 0 ; l<P ; l++)
          //implicit communication
           c[i][j] += a[i][l]*b[l][j];     }
      }
}
```

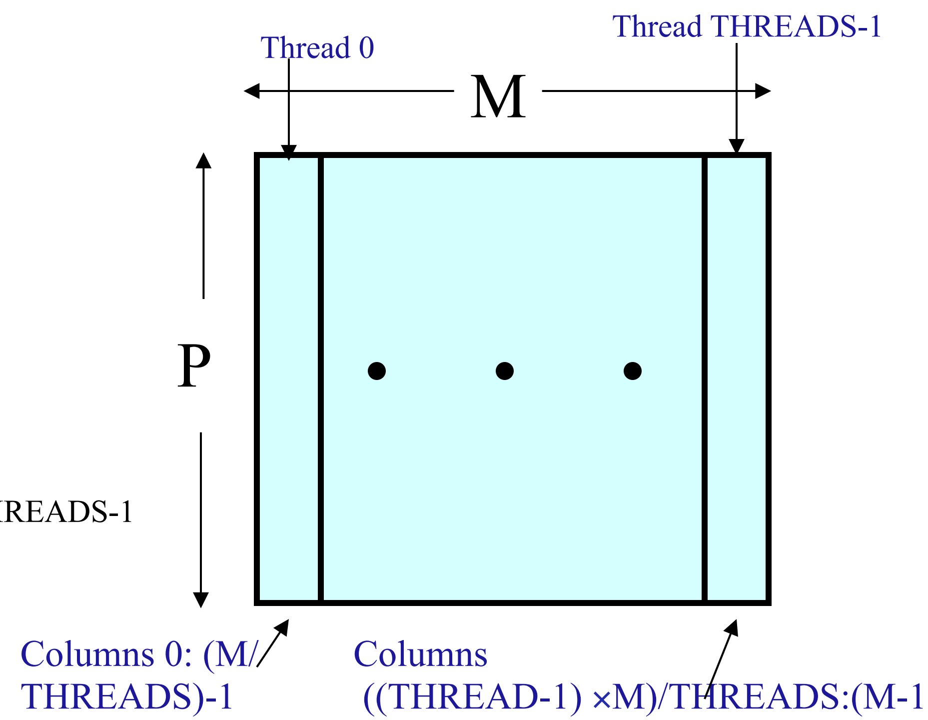
Domain Decomposition

- Exploits locality in matrix multiplication

- A (N × P) is decomposed row-wise into blocks of size (N × P) / THREADS as shown below:



- B(P × M) is decomposed column wise into M/ THREADS blocks as shown below:



•**Note:** N and M are assumed to be multiples of THREADS

Today's lecture

- Parallel Programming Languages
  - Cilk
  - UPC

Dynamic parallelism

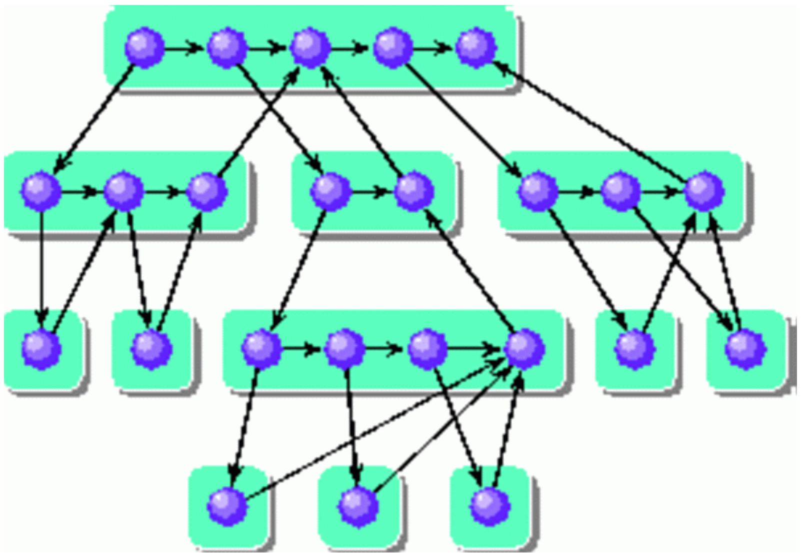
- How to support dynamic creation of parallelism, while hiding the details
- Dynamic parallelism is much harder to manage than static parallelism
  - How to keep the processors equally busy?
  - How to avoid excessive overhead costs?

Managing application complexity

- Focus on on thread-based parallelism
- Threads communicate anonymously
  - Correctness and synchronization
  - Workload distribution
- Scalability
- Task granularity

An alternative

- Let's think of a computation in terms of a graph, more precisely, a DAG
- Nodes denote computation, edges data dependence



CILK

- CILK is a programming language that supports a constrained model of thread-based parallelism with *guarantees* about *performance*
- Useful in implementing divide and conquer algorithms
- See http://supertech.lcs.mit.edu/cilk
- Cilk Plus: an extension to C and C++
  - Supported by Intel compilers and GCC 4.7
  - See http://software.intel.com/en-us/articles/intel-cilk-plus

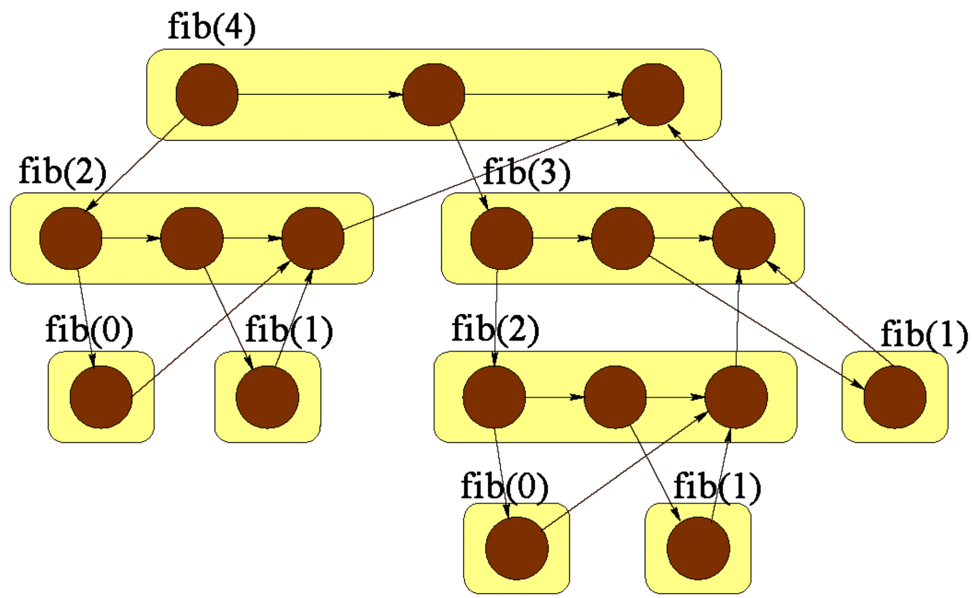
A first CILK program

- fib() is called from a dynamically spawned thread
- Non-blocking call
- Calls to fib() execute concurrently
- Parent continues until it reaches a sync barrier, and waits for children to return

```
cilk int fib (int n)
  {
    if (n < 2) return n;
    else {
      int x, y;
      x = spawn fib (n-1);
      y = spawn fib (n-2);
      sync;
      return (x+y);
    }
  }
```

Call graph for Fibonacci

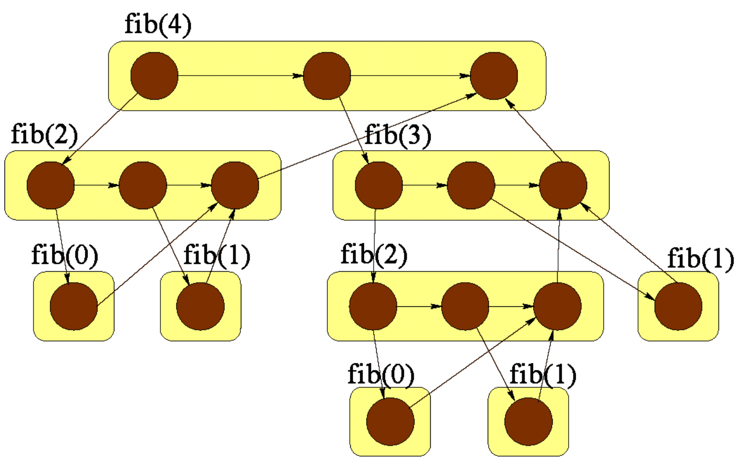
```
cilk int fib (int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```



Courtesy Alistair Dundas

A lower level CILK Model

- DAG divided into levels
- spawn(): downward edge to the next higher level
- spawn_next(): forward edge within the level
- send_argument generates an upward edge: *continuation passing*




```
Thread  fib (cont int k, int n)
{
      if (n < 2) send_argument(k,n);
      else { cont int x,y;
          spawn_next sum(k,?x,?y);
          spawn fib (x,n-1);
          spawn fib (y,n-2);
       }
}
Thread sum(cont int k, int x, int y) {
    send_argument(k,x+y);
}
```

More about the model

- Threads are non-blocking, and a parent cannot wait on a child
- Parent must spawn a successor thread to receive the return values of children
- ? variables are synchronization points and are the endpoints of an upward edge
- Send_argument transmits data along the edge and is the tail of the arrow

```
Thread  fib (cont int k, int n)
{
    if (n < 2) send_argument(k,n);
    else { cont int x,y;
        spawn_next sum(k,?x,?y);
        spawn fib (x,n-1);
        spawn fib (y,n-2);
     }
}
Thread sum(cont int k, int x, int y) {
   send_argument(k,x+y);
}
```

Work stealing

- When a processor runs out of work it *steals* work from another processor
  - Picks a processor at random
  - Removes a thread from the tail of the list of the shallowest nonempty level of the ready queue
- Why the shallowest level?
  - Ensures progress along the *critical path*
  - Granularity considerations

Performance

- Define *work* as the total time to execute the entire computation on one processor (T_1)
- *Critical path length*: the longest time to execute the threads along any dependence path (T_∞)
- Assume P processors
- Define T_P = time on P processors

Performance bounds

- $T_P \geq T_1 / P$
  - In one step, P processors can do at most P units of work
- $T_P \geq T_\infty$
  - In one step, P processors can do no more work than an infinite number of processors can
- Define the *parallelism* to be T_1 / T_∞

A greedy scheduler

- In each step, the scheduler executes as much work as it can in one step (P)
- The step is *complete* if P threads are available
- Else it is *incomplete*
- Theorem due to Graham and Brent
  - A greedy scheduler executes a computation with work T_1 and critical-path length T_∞ in time

    $$T_P \le T_1 / P + T_\infty$$

Performance

- In CILK $T_P \approx T_1 /P + c_\infty T_\infty$
- $c_\infty \approx 1.5$
- The critical path is a stronger lower bound on T_P exceeds the average parallelism T_1 /T_∞
- Otherwise, T_1 /P is the stronger bound
- Depends on the ability to have good scheduler

Matrix multiply in Cilk

- HPC Challenge (2006)

```
cilk void MM (double A[m,k], double B[k,n],
              double C[m,n],
              int m, int n, int k,
              double alpha, long columnsep)
// C +=  A*B
```

Matrix multiply in Cilk

A[m, k]
B[k, n]
C [m, n]

```
if (m+n+k<BASE) {                    /* BASE = 512 */
    cblas_dgemm(…, m, n, k, 1.0,   A, …  B, …  ..., C, …);
  } else if (m>=n && m>=k) {        /* Largest dimension is m */
    spawn MM(A,        B, C,       m/2,    n, k,  col_sep);
    spawn MM(A+m/2, B, C+m/2, m-m/2, n, k,  col_sep);
  } else if (n>=m && n>=k) {        /* Largest dimension is n */
   spawn MM(A, B,                  C,                  m, n/2,   k,  col_sep);
   spawn MM(A, B+(n/2)*col_sep, C+(n/2)*col_sep, m, n-n/2, k,  col_sep);
  } else {                            /* Largest dimension is k */
    spawn MM(A              , B,    C, m, n, k/2,   col_sep);
    // Store into another variable then add, or sync.
    sync;
    spawn MM(A+(k/2)*col_sep, B+k/2, C, m, n, k-k/2,  col_sep);
  }
```

Fin