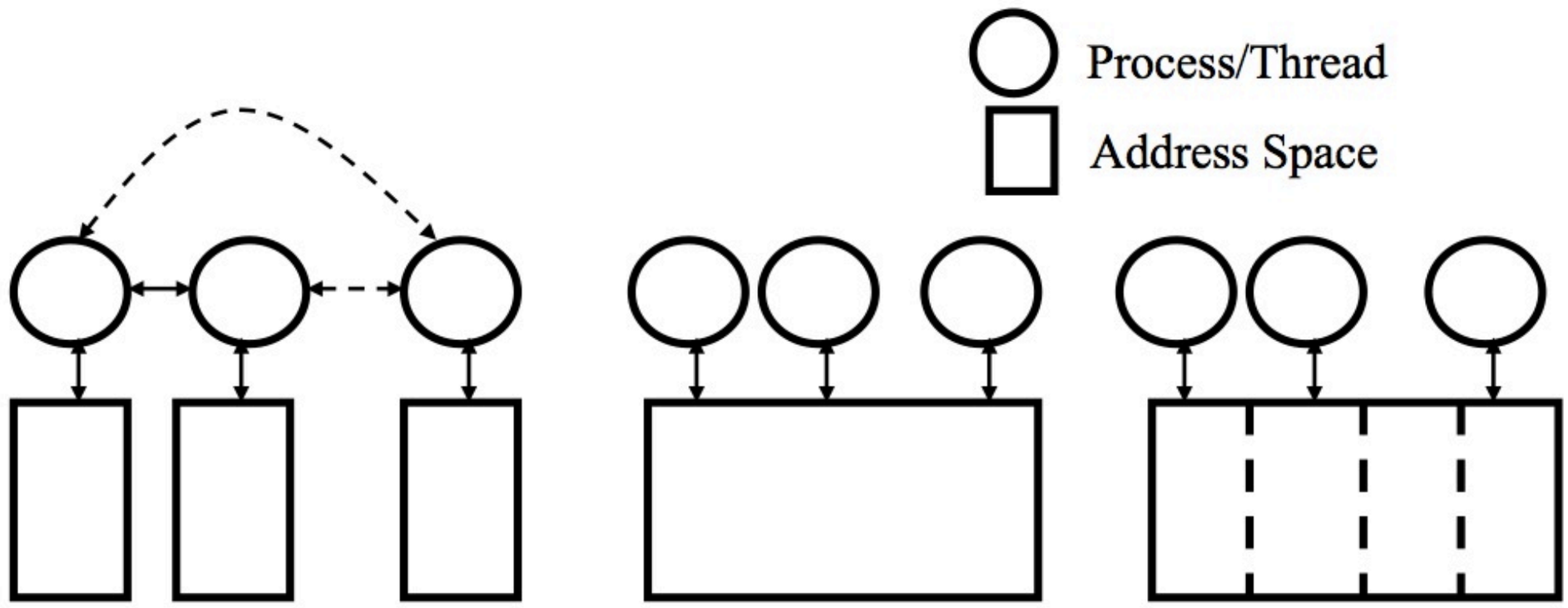
UPC | Cilk | OpenMP

androwis abumoussa

background

programming models



message passing shared memory dsm

introductions

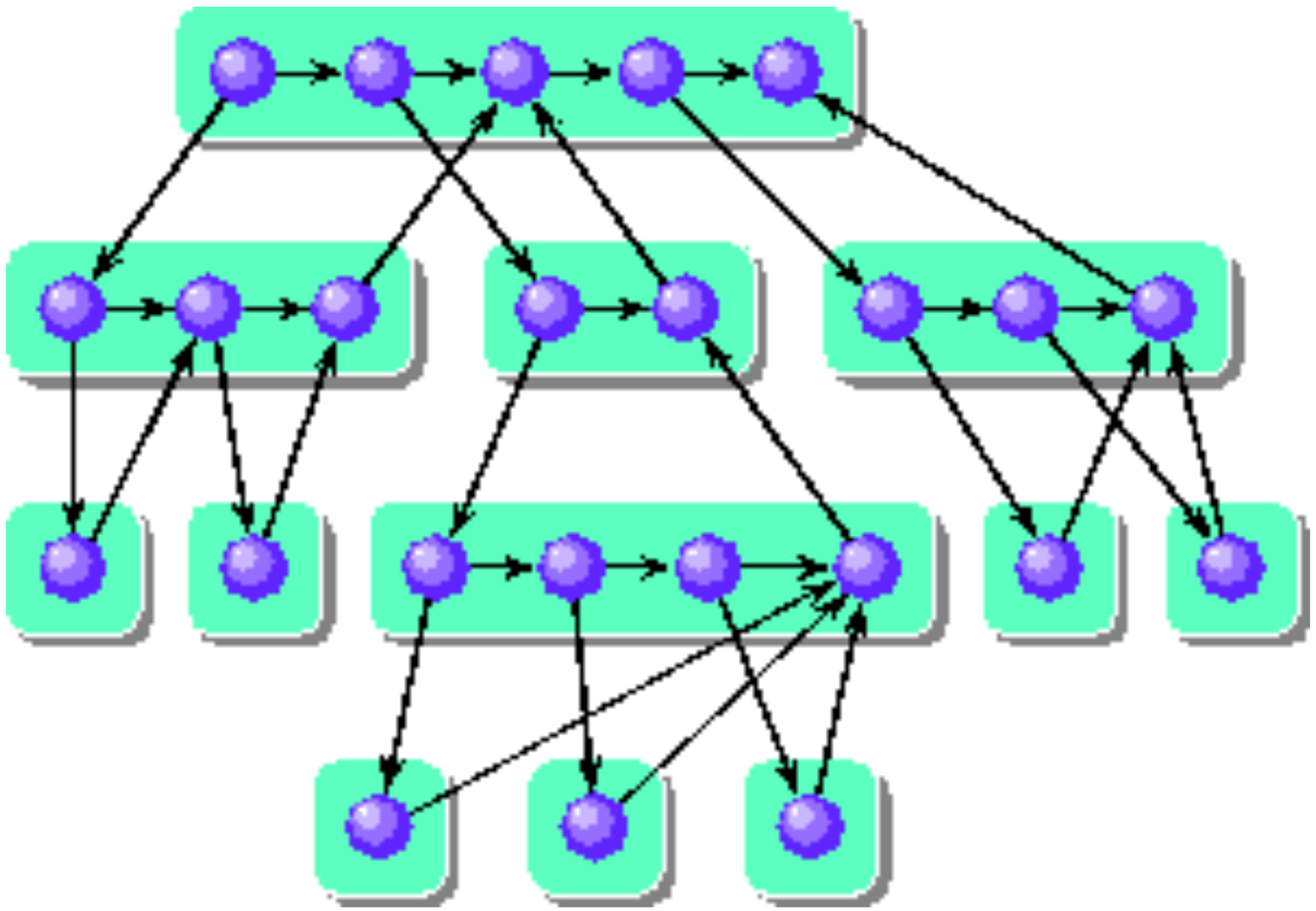
Cilk : compiler directives, library routines

OpenMP : compiler directives, library routines, & environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.

Unified Parallel C : a parallel extension of the C programming language intended for multiprocessors w/ a common address space


cilk
from MIT

cilk



cilk

Goal : simplify writing multithreaded code

5 new keywords

Shared Memory

- programmers id code that can safely be executed in parallel
  - Nested loops > data-parallelism > cilk threads
  - Divide-&-conquer algorithms > task parallelism > cilk threads

- run-time env decides how to divide the work between processors
  - can run without rewriting for any sp number of processors

cilk : keywords

cilk - procedure, capable of being spawned (main needs to have cilk)

spawn - child cilk procedures which execute in parallel w/ parent

sync - wait until all children return results to parent frame (barrier)

inlet

abort

cilk : hello world

```
#include <cilk-lib.cilkh>

cilk void hello(int i)
{
  printf("Hello World from thread %d\n",i);
}

cilk int main(int argc, char *argv[])
{
      int n, i;

      if (argc != 2) {
         fprintf(stderr, "Usage: hello [<cilk options>] <n>\n");
         Cilk_exit(1);
      }

      n = atoi(argv[1]);

      for (i=0;i<n;i++)
        spawn hello(i);
      sync;

      return 0;
}
```

cilk : inlets & abort

```
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);
    }
}
int max, ix = -1;
inlet void update(int val, int i){
      if(idx==-1 || val > max){
         ix = i; max = val;
      if(max == 100){
        abort; //existing threads
      }

cilk int product (int *a, int n)
{
    for(i =0; i<1000000; i++)
        update (spawn foo(i),i);
}
```

Communication b/w threads normally occurs by storing spawned function result into variable in parent's frame.

The alternative is to use an inlet. An inlet is a function internal to a Cilk procedure which handles the results of a spawned procedure call as they return. One major reason to use inlets is that all the inlets of a procedure are guaranteed to operate atomically with regards to each other and to the parent procedure.

inlet - ids function defined w/in a procedure as inlet.

abort - can only be used inside an inlet; it tells the scheduler that any other procedures that have been spawned off by the parent procedure can safely be aborted.

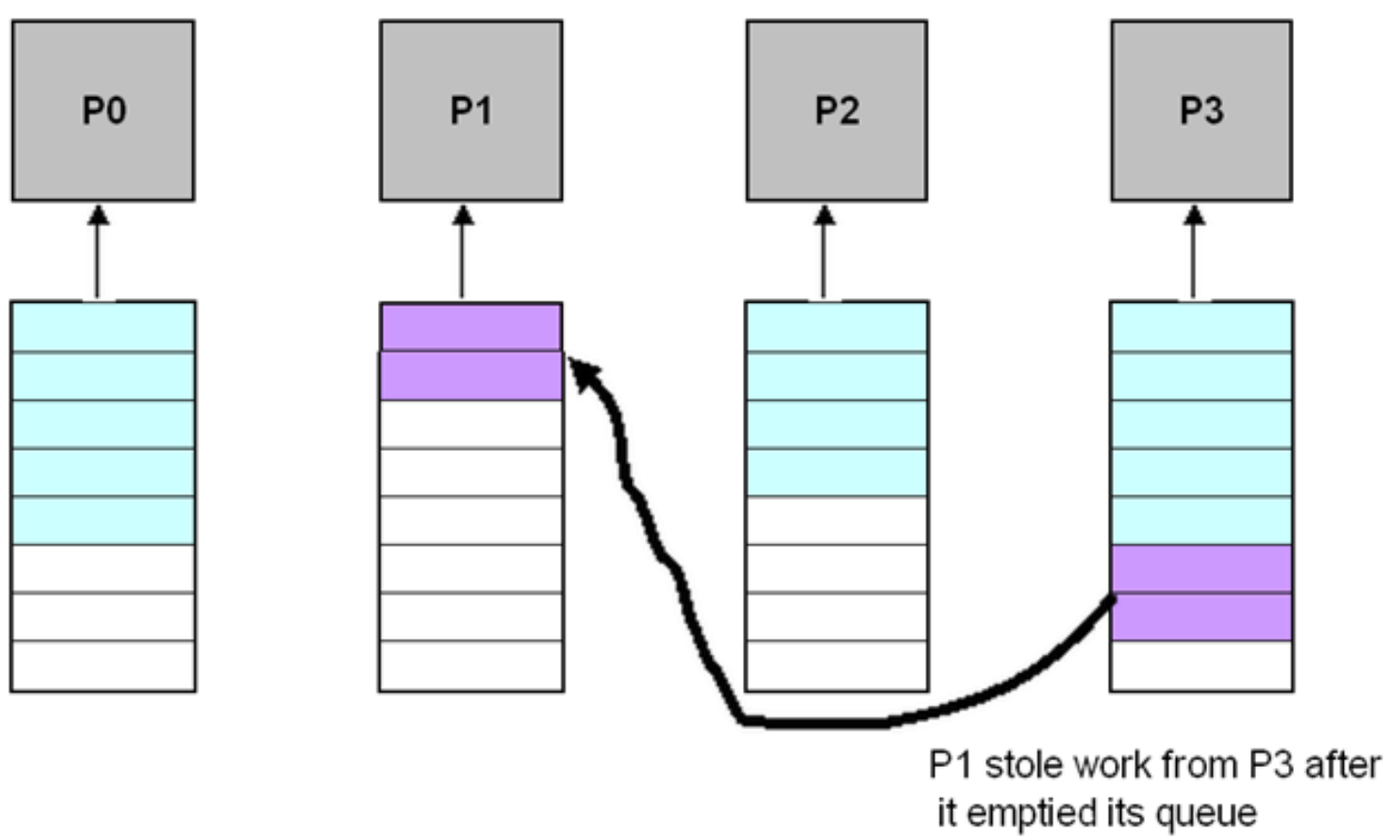
cilk : scheduler

- work-first scheduling strategy coupled with a randomized work stealing load balancing strategy shown to be optimal
- cilk scheduler maps cilk threads onto processors dynamically at run-time
- greedy scheduling - do as much work as possible
- >= P threads as ready, pick one and run

cilk : scheduler

Work Stealing

The processor maintains a stack, places each frame that it has to suspend in order to handle a procedure call on that stack along w/ the state.



cilk : global sync

cilk provides locks

```
#include <cilk-lib.h>
cilk_lockvar mylock;
{
   cilk_lock_init(mylock);

   Cilk_lock(mylock);  //enter critical section
    ...
   Cilk_unlock(mylock);// exit critical section
}
```

cilk : pointers

- has support for stacks and heaps
- parent pointers can be passed to children, but not vice versa

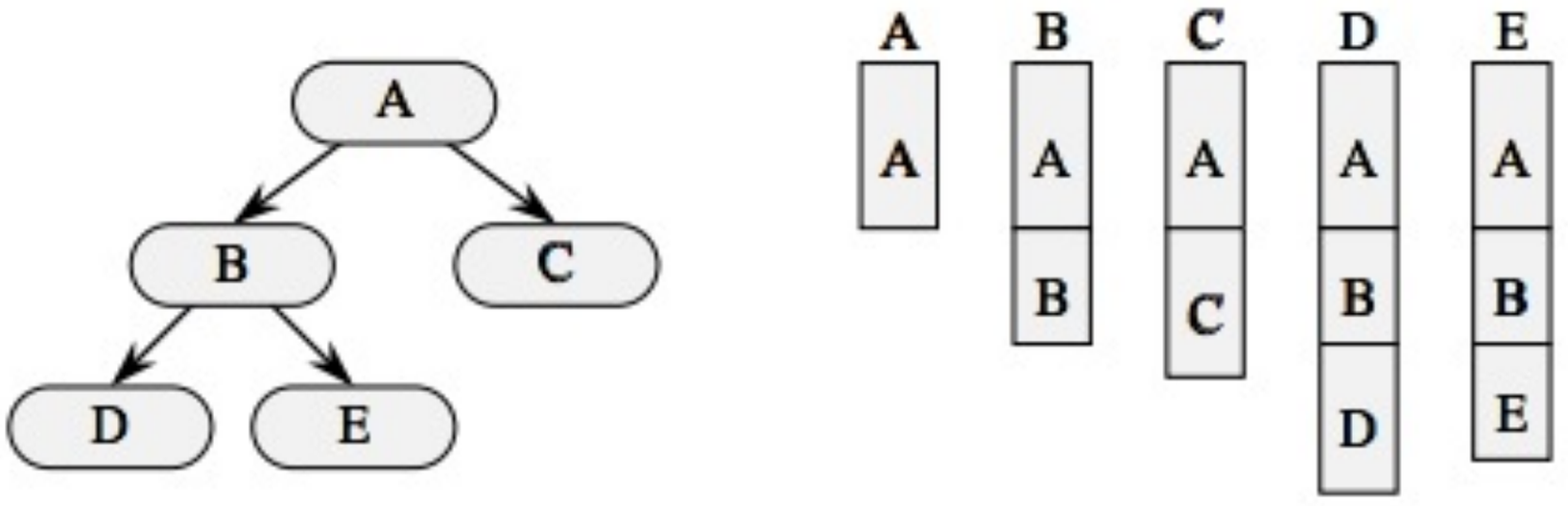
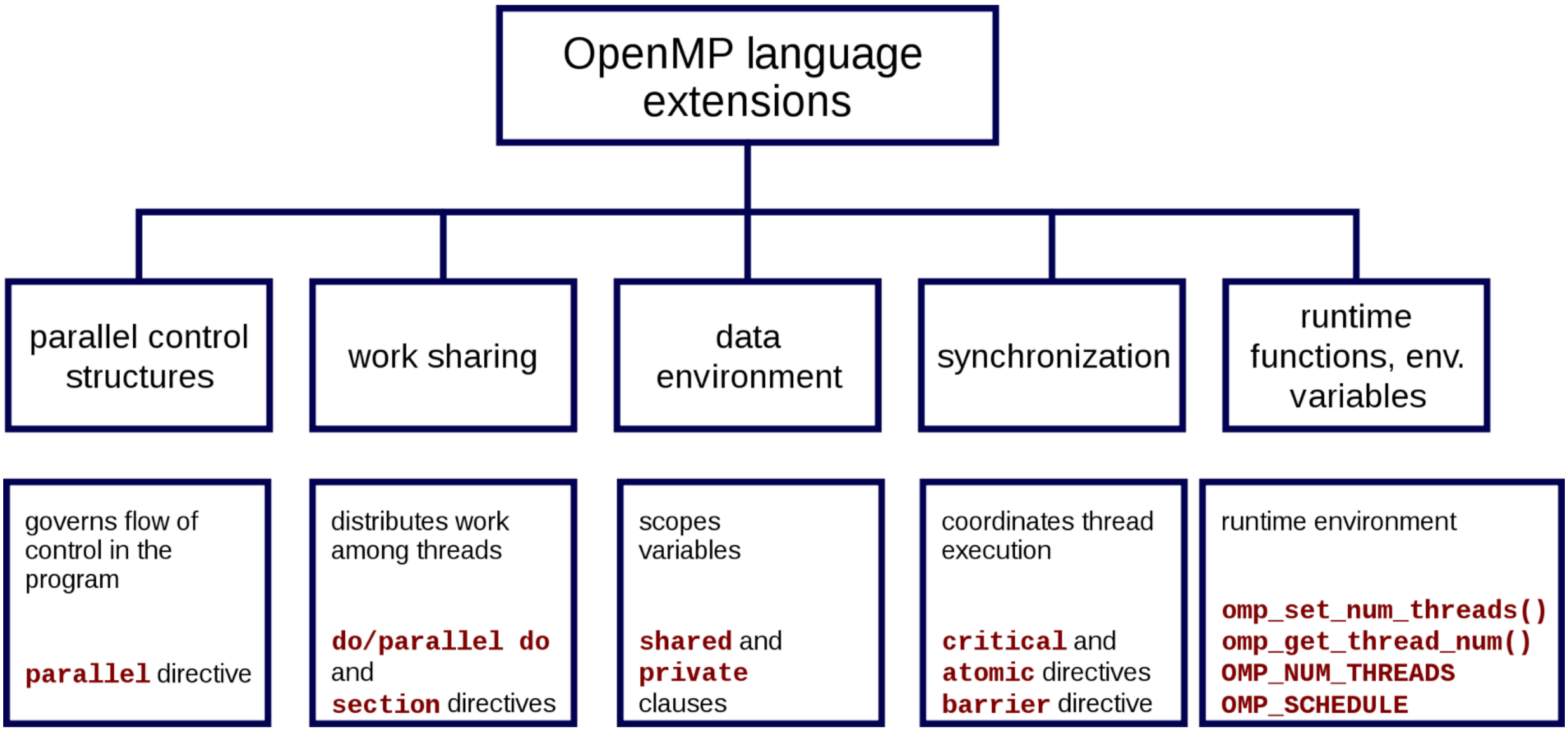


Figure 2.5: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

OpenMP

open multi processing



OpenMP

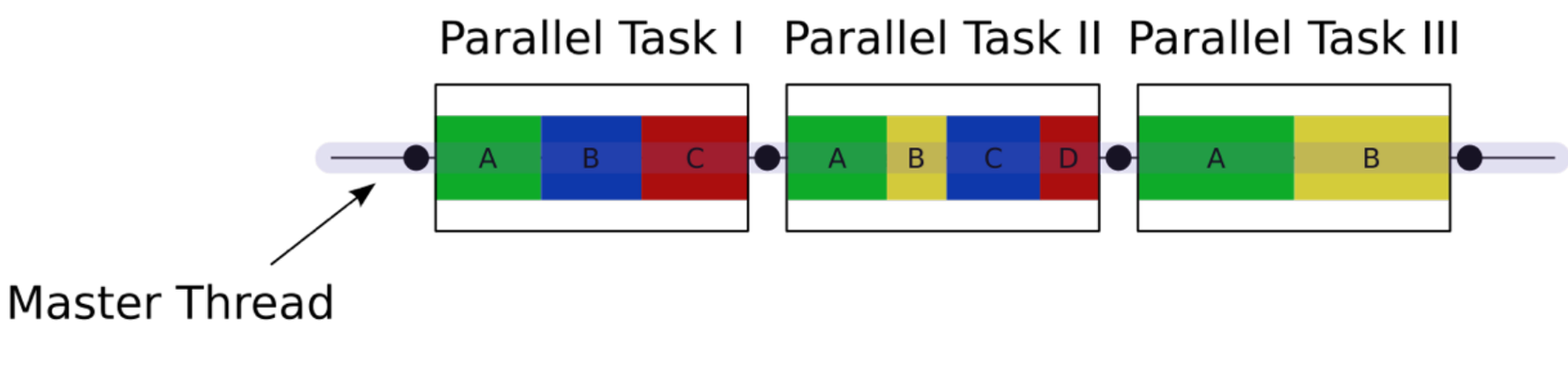
a multi threading implementation where a master thread forks a specified number of slave threads and a task is divided among them.

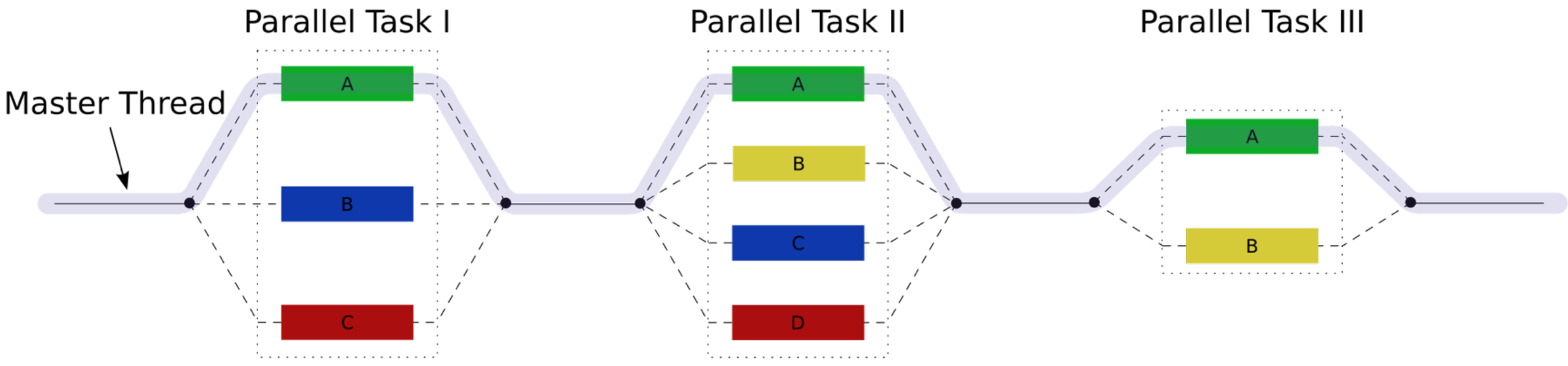
- parallel section marked by preprocessor directives that cause threads to be created

- threads assigned to processors at runtime

- shared address model (communication by sharing variables)

OpenMP


Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
A
B
C
D
A
B
Master Thread


Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
A
B
C
D
A
B

OpenMP : threads

The pragma ***omp parallel*** is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

sample code

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
    printf("Hello, world.\n");
  return 0;
}
```

compile with flags

```
$gcc -fopenmp hello.c -o hello
```

output on a dual core

```
Hello, world.   Hello, wHello, woorld.
Hello, world.   rld.
```

OpenMP : work sharing

how to assign independent work to one or all of the threads.

omp for or ***omp do***:
used to split up loop iterations among the threads (loop constructs).

sections:
assigning consecutive but independent code blocks to different threads

single:
specifying a code block that is executed by only one thread, a barrier is implied in the end

master:
similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

OpenMP : work sharing

Example: init values of a large array in parallel, using each thread to do part of the work

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

OpenMP : clauses

OpenMP is a shared memory programming model so variables in OMP code are visible to all threads by default.

OpenMP provides private variables necessary to avoid race conditions and can pass values between the sequential part and the parallel region (the code block executed in parallel)

OpenMP : consistency

- ***shared***: parallel region data visible by all threads
- ***private***: parallel region data private to each thread
  - a local temp variable copy
  - can't reuse in non-parallel region

- ***firstprivate***: like *private* except initialized to original value.
- ***lastprivate***: private data value copied to a global variable using the same name outside the parallel region iff current iteration is last iteration in the parallelized loop

- ***reduction***: a safe way of joining work from all threads after construct.

OpenMP : syncing

- **critical** : thread-wise mutual exclusion
- **atomic** : mutual exclusion on memory updates
- **barrier** : wait for all threads to reach
- **ordered** : as if it is sequential
- flush
- locks

OpenMP : data example

```
#include <omp.h>
#define NUM_THREADS 4
#define NUM_START 1
#define NUM_END 10

int main() {
   int i, nRet = 0, nSum = 0, nStart = NUM_START, nEnd = NUM_END;
   int nThreads = 0, nTmp = nStart + nEnd;
   unsigned uTmp = (unsigned((abs(nStart - nEnd) + 1)) *  unsigned(abs(nTmp))) / 2;
   int nSumCalc = uTmp;
   if (nTmp < 0)
      nSumCalc = -nSumCalc;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel default(none) private(i) shared(nSum, nThreads, nStart, nEnd)
   {
      #pragma omp master
      nThreads = omp_get_num_threads();

      #pragma omp for
      for (i=nStart; i<=nEnd; ++i) {
            #pragma omp atomic
            nSum += i;
      }
   }
}
```

OpenMP : reduction

reduction(operator | intrinsic : list): Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the datatype, then they increments the datatype

```
#define N 10000 /*size of a*/
/*some function that populates the elements of a*/
void calculate(long *){...};

int i;
long w, sum = 0;
long a[N];
calculate(a);

/*forks off the threads and starts the work-sharing construct*/
#pragma omp parallel for private(w) reduction(+:sum) schedule(static,1)
for(i = 0; i < N; i++)
   {
     w = i*i;
     sum = sum + w*a[i];
   }
printf("\n %li",sum);
```

OpenMP : pros

- simple : no need to deal w Message Passing
- data layout & decomposition handled by directives
- incremental parallelism
- unified code when no OpenMP support

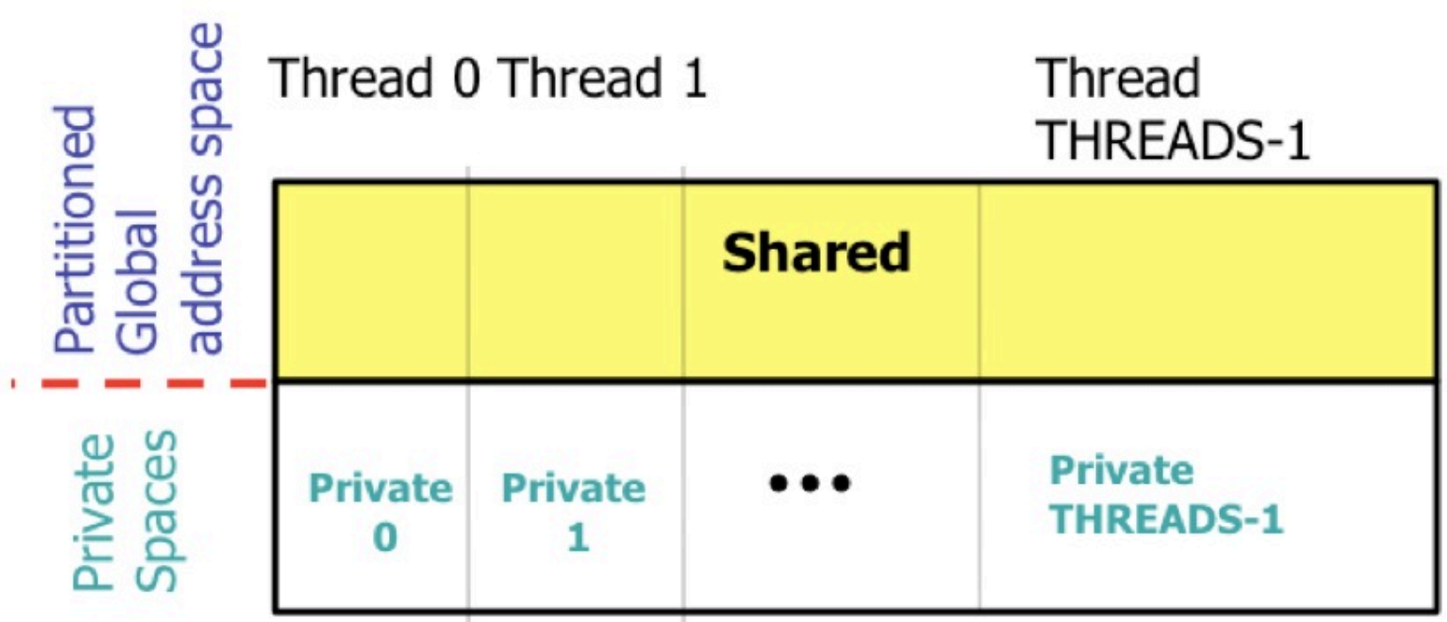
OpenMP : cons

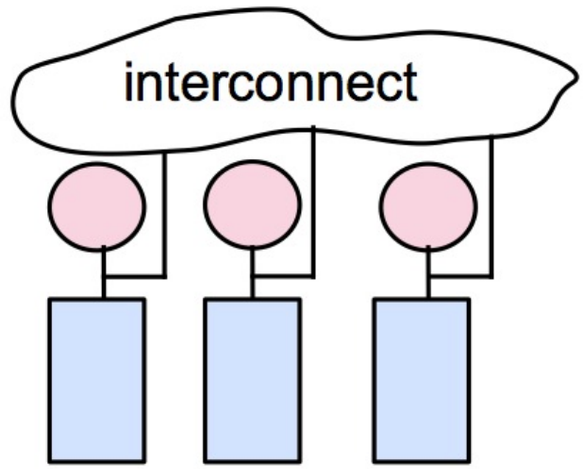
- only runs efficiently in shared memory
- compiler has to support OpenMP
- scalability determined by memory
- no compare-and-swap
- can’t use GPU
- multithreaded executables have long startup times

UPC

Unified Parallel C

upc : distributed mem





Goals : remove friction of data movement & synchronization

- co-locate data w/ processes
- aggregate multiple accesses to remote data
- overlap communication w/ computation

upc : api features

- partitioned global address space
  - (part of shared data co-located w/ each thread)
- threads created at launch (and bound to a cpu)
- a memory model
- sync primitives (barriers / locks / load-store)

Unified Parallel C

A partitioned shared memory parallel programming language

1) provide efficient access to underlying machine (manages shared/private)

2) establish common syntax and semantics for parallelism

3) tries to minimize the overhead involved in communication b/w threads

Global Addressable Space Network

"A portable high-performance communication layer for global address-space languages"

- Provide a global shared memory abstraction to the user, regardless of the hardware implementation
- Make distinction between local & remote memory explicit
- Get the ease of shared memory programming, and the performance of message passing
- One-sided communication (GET/PUT) simpler than msg passing
- Programmer has control over performance-critical factors
    - data distribution (lacking in OpenMP)

Unified Parallel C

A collection of threads operating in a single global address space, which is logically partitioned among threads. Each thread has affinity with a portion of the globally shared address space. Each thread also has a private space.

upc : keywords

- **shared** : type qualifier :: data shared b/w all threads.

```
shared int y[THREADS];          /* one y on each thread */

shared int a[100][THREADS];     /* one a[100] on each thread*/

shared int b[100][12*THREADS];  /* one copy of b[100][12] on each thread */

shared int x;                   /* one x on entire system */
```

upc data layout

Shared objects placed in memory based on affinity

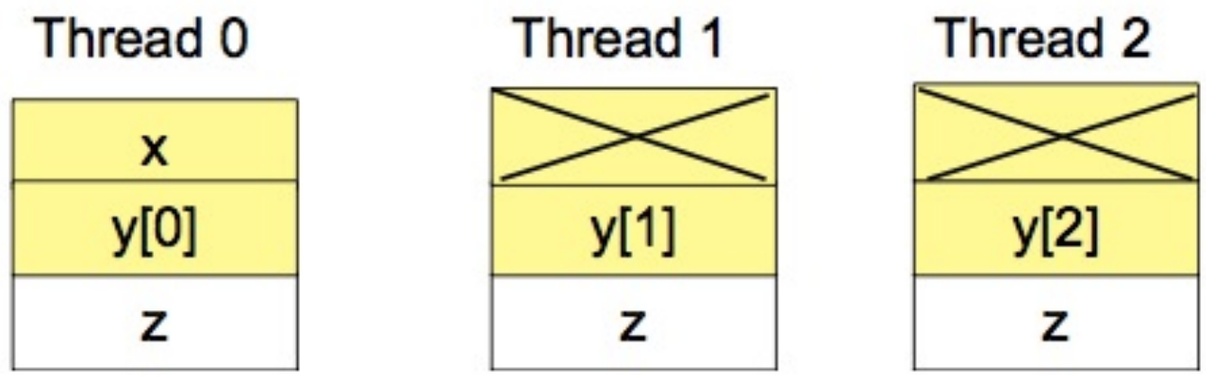
• Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer

• All non-array shared qualified objects ,i.e. shared scalars, have affinity to thread 0

• Threads access shared and private data

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

For THREADS = 3, we get the following layout



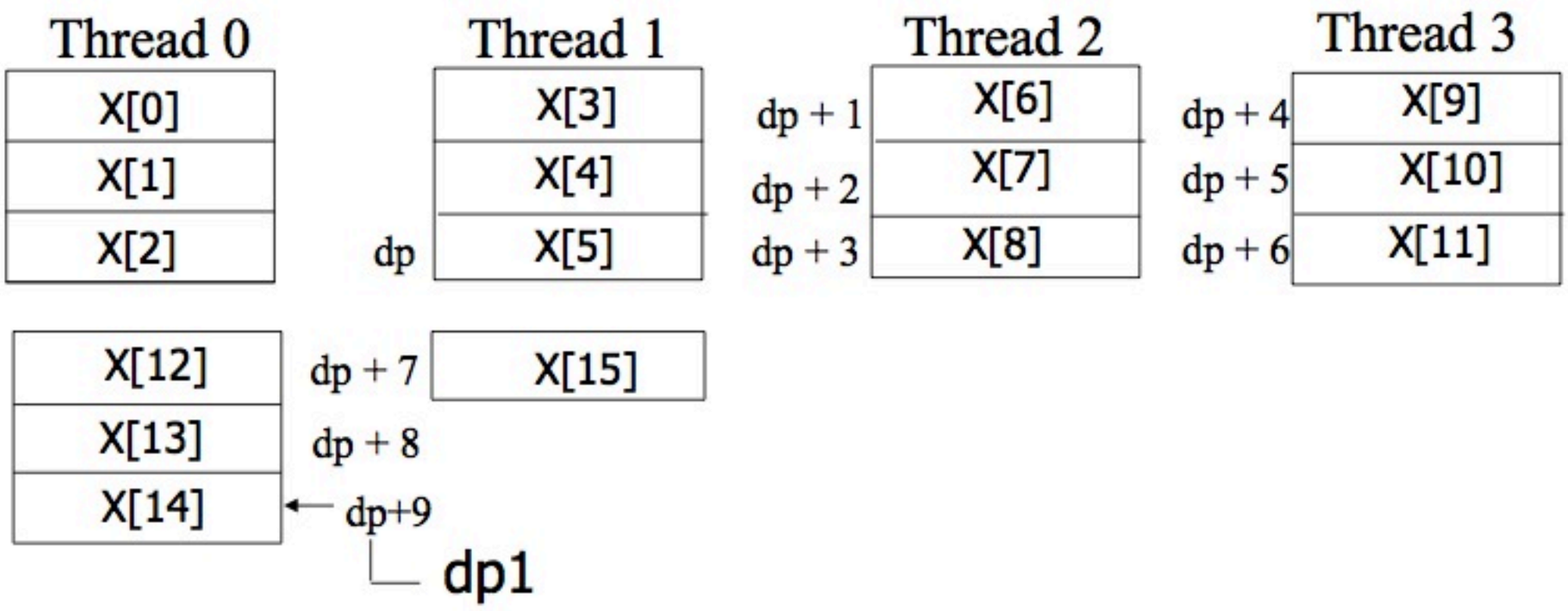
upc: pointers

- Internally, pointers to shared objects have three logically separate components: a thread number, a local address, phase.
- The thread number is used to determine where the remote reference is to be done, and the local address is used on that thread as if it were in that thread's "local" view.

```
 int *p;              /* a local item which points locally */
shared int *p;    /* a local item which points elsewhere */
int *shared p;    /* (not recommended) a shared item which points elsewhere */
shared int *shared p;  /* access (all threads) to data in shared space */
```

example thread addition

```
// Assume THREADS = 4

shared[3] int x[N], *dp=&x[5], *dp1;

dp1 = dp+9;
```



upc : upc_forall

```
 // Assume THREADS = 4
shared int a[100],b[100],c[100];
int i;
upc_forall(i=0; i<100; i++; (i*THREADS)/100)
                    a[i] = b[i] * c[i];
```

Distributes independent iterations across threads

Simple C-like syntax and semantics
—upc_forall(init; test; loop; affinity)

Affinity is used to enable locality control
—usually, map iteration to thread where the iteration's data resides

Affinity can be
—an integer expression, or a
—reference to (address of) a shared object

upc : upc_forall

```
// Assume THREADS = 4

shared int a[100],b[100],c[100];

int i;

upc_forall(i=0; i<100; i++; (i*THREADS)/100)

    a[i] = b[i] * c[i];
```

i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3

upc : consistency model

- memory reference can be annotated (strict/relaxed)
- strict : implies sequential
- relaxed: implies “local consistency model” - for a local thread, all accesses to a shared reference occur in the order they were sent

upc : tasks

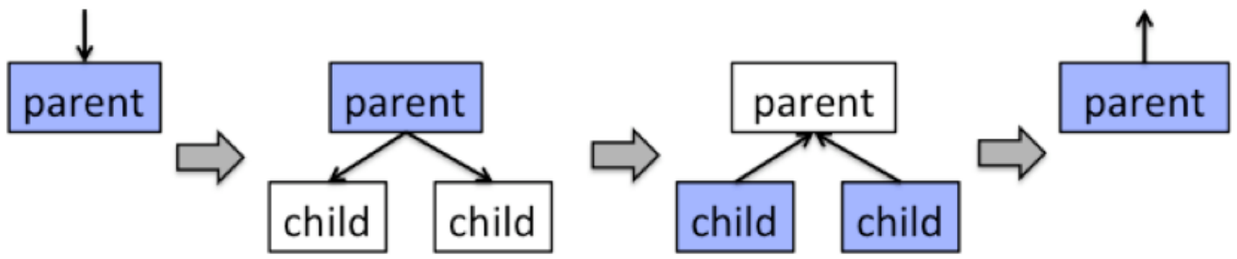
- library that helps users perform dynamic load balancing
- unit of executable code, pointers to in/outputs

```
task_func(void *input, void *output); // Define a task.
```

```
/*task queues api*/

taskq_put
taskq_execute
taskq_steal
taskq_isEmpty
taskq_all_isEmpty
taskq_all_run
```

```
/*task synchronization*/

void taskq_wait (taskq_t *taskq);
void taskq_fence (taskq_t *taskq);
```

- http://upc.lbl.gov/task.shtml



upc: global sync

- 2 methods : barriers & forall loops
  - upc_notify(), upc_wait(), upc_barrier
  - forall : The forall statement allows programmers to describe a global loop with assignment of threads to loop indices made by the compiler and/or runtime system

upc : global sync

`ALLSYNC` is the most “expensive” because it provides barrier-like synchronization.

`NOSYNC` is the most “dangerous” but it is almost free.

`MYSYNC` provides synchronization only between threads which need it.

Examples

Sequential version

```
int fib (int n) {

  if (n < 2) return 1;
  else {
    int rst = 0;
    rst += fib (n-1);
    rst += fib (n-2);
    return rst;
  }
}
```

Cilk version

```
cilk int fib (int n) {
 if (n < 2) return 1;
 else {
   int rst = 0;
   rst += spawn fib (n-1);
   rst += spawn fib (n-2);
   sync;
   return rst;
   }
}
```

Sequential version

```
int fib (int n) {

  if (n < 2) return 1;
  else {
    int rst = 0;
    rst += fib (n-1);
    rst += fib (n-2);
    return rst;
  }
}
```

OpenMP version

```
int fib (int n) {

  if (n<2) return 1;
  else {
    int rst = 0;

  #pragma omp task{
     #pragma omp atomic
      rst += fib(n-1);
    }
   #pragma omp task{
     #pragma omp atomic
      rst += fib(n-2);}
    }

   #pragma omp taskwait
   return rst;
}
```

Sequential version

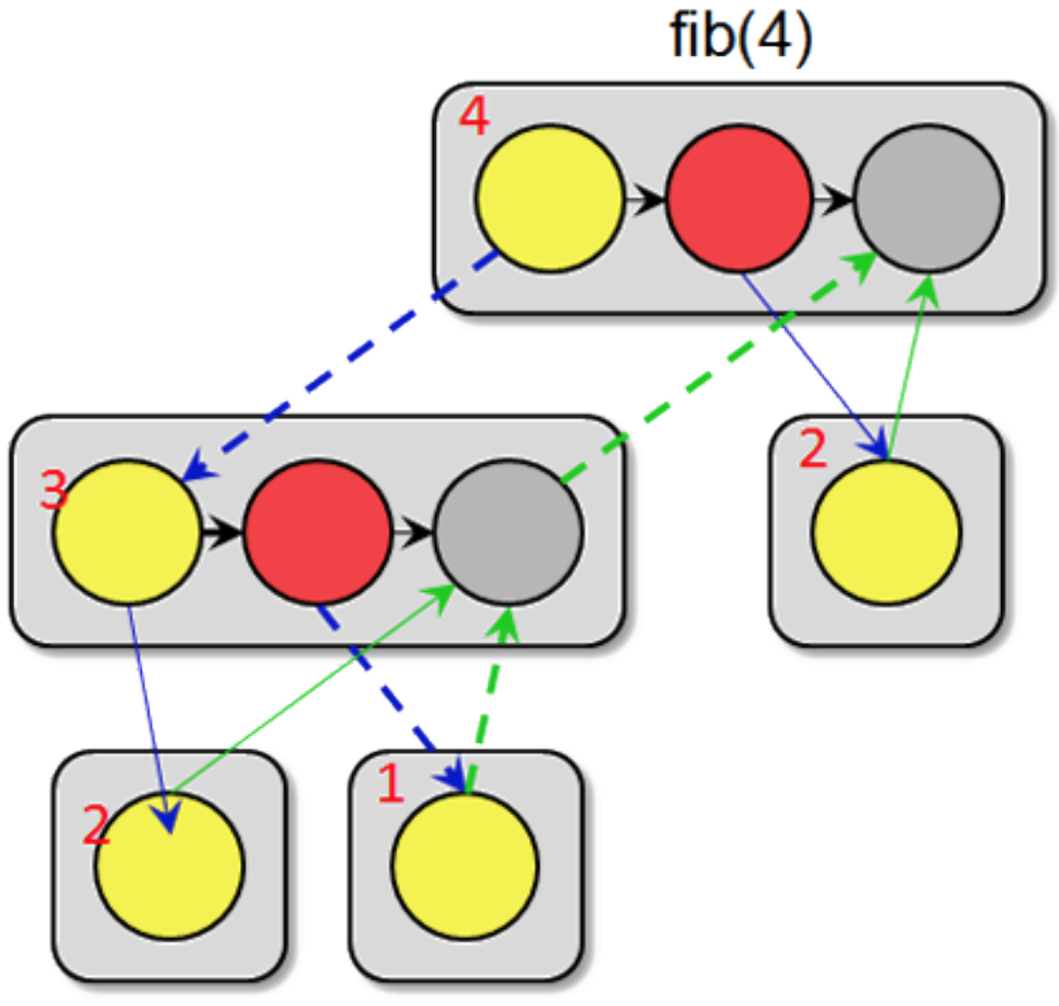
```
int fib (int n) {

  if (n < 2) return 1;
  else {
    int rst = 0;
    rst += fib (n-1);
    rst += fib (n-2);
    return rst;
  }
}
```

UPC version

```
#include "upc.h"
#include "upc_task.h"

taskq_t *taskq;

void FIB( int *n, int *out ) {
 int n1 = *n-1;
 int n2 = *n-2;
 int x, y;

 /* termination condition */
 if (*n < 2){
  *out = *n; return;
 }

 taskq_put(taskq, FIB, &n1, &x);
 taskq_put(taskq, FIB, &n2, &y);
 taskq_wait(taskq);

 *out = x + y;
}

int main(int argc, char *argv[]){
  int N, result;
  N = (int) atoi(argv[1]);
  taskq = taskq_all_alloc(1, FIB, sizeof(int), sizeof(int));

  upc_barrier;

  if (MYTHREAD==0)
   taskq_put(taskq, FIB, &N, &result);

  /* executes tasks in the taskq */
  taskq_all_run(taskq);

  taskq_all_free(taskq);
}
```

cilk

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



Work (W)	= 9
Span/Depth (D)	= 6
Parallelism = W/D	= 1.5

bibliography

- Blumofe, Robert D., et al. Cilk: An efficient multithreaded runtime system. Vol. 30. No. 8. ACM, 1995.
- Randall, Keith H. Cilk: E cient Multithreaded Computing. Diss. Massachusetts Institute of Technology, 1998.
- Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.
- Unified Parallel, C. "URL: http://en. wikipedia. org/wiki." Unified Parallel C.
- openmp.org/mp-documents/omp-hands-on-SC08.pdf
- www.cs.princeton.edu/courses/archive/fall10/.../docs/Cilk.pdf
- faculty.knox.edu/dbunde/teaching/cilk/Share
- www.cs.cornell.edu/~bindel/class/cs5220-f11/slides/lec10.pdf
- Berkeley Unified Parallel C (UPC) Project upc.lbl.gov