Dynamic Allocation of Data and Work - it might be hard or impossible to statically assign an even workload for several reasons:

- server processes client requests - requests determine the amount of work
- dynamic work created on the fly as the computation proceeds
- size of static data does not reflect amount of computation (i.e., hard to load balance)

Work Queue - data structure for dynamically assigning work to threads or processes. A thread/process *producing* extra work puts it into the work queue. An idle thread/process remove (*consumes*) work from the queue to keep busy.

Types of work queues might depend on the problem being solved. Some options:

- FIFO queue - add new items to rear and remove items from the front
- LIFO stack - add new items to the top of the stack and remove items from the top (e.g., depth-first search)
- randomized queue - remove items randomly from queue
- priority queue - each item has a priority associated with it. Remove the item with highest priority (e.g, best-first search)

1. Textbook example: Collatz Conjecture ("For any positive integer a_0, does the process defined by:

$$a_i = \begin{cases} 3a_{i-1} + 1 & \text{if } a_{i-1} \text{ is odd} \\ a_{i-1}/2 & \text{if } a_{i-1} \text{ is even} \end{cases}$$ For example, a_0 =17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Problem of interest: What is the largest factor of expansion, $\max(a_i)/a_0$? For a_0 =17 it is 52/17 = 3.06.
Note: the work queue, *q*, is initialed with first P positive integers

```
 1  float ef=1.0;                          Best expansion factor
 2  int bestA=1;                           Record which a_i had best expansion
 3  int head=0;                            Next item on Q to process
 4  forall(index in(0 .. P-1))             Define P threads to perform test
 5  {
 6    int a=1;                             Test number
 7    float myEF=1.0;                      Local Expansion Factor
 8    int big, atest=1;                    Locals to compute expansion factor
 9
10    while(a<runSize)                     Limit to 2 billion or expand single precision
11    {
12      exclusive                          Get a from Q and leave a + P
13      {
14        a=q[head];
15        q[head]=a+P;
16        head=(head+1)%P;
17      }                                  End of single thread section
18      atest=a;                           Set up for a test
19      big=a;                             The start could be the largest value we see
20      while(atest!=1)
21      {
22        if(even(atest))
23        {
24          atest=atest/2;
25        }
26        else
27        {
28          atest=3*atest+1;
29          big=max(big, atest);
30        }
31      }
32      myEF=big/a;                        Compute expansion for this a
33      exclusive                          Get a from Q and leave a + P
34      {
35        if(myEF>ef)                      Record any progress (after 1st time)
36        {
37          ef=myEF;
38          bestA=a;
39        }
40      }
41    }
42  }
```

a) What is purpose of the first `exclusive` statements?

b) What is purpose of the second `exclusive` statements?

c) Would it have been better to give each thread runSize/P integers statically?

2. Using a work queue, the granularity of work presents what tradeoffs:

- small-grain size of work

- large-grain size of work

3. What is the problem of having a single work queue when P is large?

4. If we have multiple work queues, what are the advantage(s) and disadvantage(s) of each of the following:

a) each thread has its own work queue

b) a group of threads share a work queue with several disjoint groups

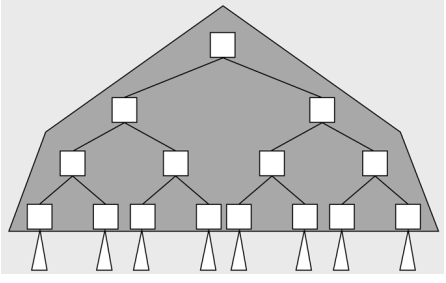
c) a group of threads share a work queue with several disjoint groups, plus a global work queue where local queues can retrieve chucks of work from if the local queue becomes empty, or deposit chucks of work to if the local queue becomes full. (*Hoard allocation*)

5. Distributing Tree Data Structure across multiple threads/processes is challenging because:

- trees are typically constructed using memory pointers which are only valid with a shared memory
- dynamic nature of tree implies extensive-performance limiting communication
- irregular structure of some trees makes it difficult to reason about communication and load-balance.

Trees are too useful to ignore!

Allocation by sub-tree: Cap - duplicate top of tree at each process. Figure 5-18 with P = 8.



a) How does a cap help?

b) When would a cap not help?

Chapter 6. Programming with Threads - shared memory machine (mult-core or SMP computers)

6. What is a thread?

POSIX Threads functions:

pthread_create()

```
int pthread_create(                     // create a new thread
  pthread_t *tid,                       // thread ID
  const pthread_attr_t *attr,           // thread attributes
  void *(*start_routine)(void *),       // pointer to function to execute
  void *arg                             // argument to function
);
```

Arguments:

- The thread ID of the successfully created thread.
- The thread's attributes, explained below; the `NULL` value specifies default attributes.
- The function that the new thread will execute once it is created.
- An argument passed to the `start_routine()`.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

Use a structure to pass multiple arguments to the start routine.

pthread_join()

```
int pthread_join(                       // wait for a thread to terminate
  pthread_t tid,                        // thread ID to wait for
  void **status                         // exit status
);
```

Arguments:

- The ID of the thread to wait for.
- The completion status of the exiting thread will be copied into `*status` unless `status` is `NULL`, in which case the completion status is not copied.

Return value:

0 for success. Error code from `<errno.h>` otherwise.

Notes:

Once a thread is joined, the thread no longer exists, its thread ID is no longer valid, and it cannot be joined with any other thread.

pthread_self()

```
pthread_t pthread_self();               // Get my thread ID
```

Return value:

The ID of the thread that called this function.

pthread_equal()

```
int pthread_equal(                      // Test for equality
  pthread_t t1,                         // First operand thread ID
  pthread_t t2                          // Second operand thread ID
);
```

Arguments:

Two thread IDs

Return value:

- Nonzero if the two thread IDs are the same (following the C convention).
- 0 if the two threads are different.

void pthread_exit()

```
void pthread_exit(                      // terminate a thread
  void *status                          // completion status
);
```

Arguments:

The completion status of the thread that has exited. This pointer value is available to other threads.

Return value:

None.

Notes:

When a thread exits by simply returning from the start routine, the thread's completion status is set to the start routine's return value.

Thread Attributes

```
pthread_attr_t attr;                    // Declare a thread attribute
pthread_t tid;
pthread_attr_init(&attr);               // Initialize a thread attribute
pthread_attr_setdetachstate(&attr,      // Set the thread attribute
        PTHREAD_CREATE_UNDETACHED);
pthread_create(&tid, &attr, start_func, NULL);  // Use the attribute
                                                // to create a thread
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);            // Destroy the thread attribute
```

Notes:

There are many other thread attributes. See a POSIX Threads manual for details.

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;                  // Declare a pointer to a lock
lock=(pthread_mutex_lock_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
   /*
    * Code that uses this lock.
    */
pthread_mutex_destroy(lock);
free(lock);
```