## Threading Building Block (TBB) by Intel

- C++ library
  - Represents higher-level task-based parallelism that abstracts platform details and threading from the user
  - Doesn't require special language/compiler
  - Can be used on any processor with any OS
- Task scheduler implements task stealing for dynamic workload balancing
- Implements common parallel performance pattern such as parallel loops, flow graphs etc.

## TBB

- Implements concurrent containers – scalable alternative to serial data containers such as C++ STL containers
- Provides a comprehensive set of synchronization primitives – mutexes, condition variables, atomic operations
- Provides the ability to specify priorities for tasks/ task groups
- Implements parallel data structures such as concurrent priority queues, concurrent unordered sets, etc.

## Grand Central Dispatch (GCD) by Apple

- Also implements task queues using a thread pool paradigm
  - Uses threads at the lower level
- Applications create tasks that can be expressed either as a function or as a "block" (akin to a closure)
- Mac OS X, Apache HTTP Server

## GCD

- Application framework
  - Dispatch queues – queue of tasks; concurrent or serial. Queues with different priority levels are created by the library that can execute tasks concurrently
  - Dispatch source – allow client to register blocks/functions to execute asynchronously on system events or POSIX signal
  - Dispatch groups – allow several tasks to be grouped for later joining
  - Dispatch semaphores – allow a client to permit a certain number of tasks to execute concurrently

# Lithe: Composing Parallel Software Efficiently

Pan, Hindman, Asanovic
MIT/Berkeley
PLDI 2010

## Software Trends

- Parallel applications using parallel libraries in turn using other parallel libraries
- E.g.
  - SuiteSparseQR
    - Incorporated into MATLAB
    - Uses Intel's TBB
    - Calls BLAS libraries
      - Which use OpenMP
- State of the art
  - User required to make decisions on degree of parallelism at each level
  - Thread virtualization implies user has no control over influencing resource allocation

## Possible Solutions

- Require all parallelism to be expressed using a universal high-level abstraction
- Provide a low-level substrate for parallel libraries to interface with

## Lithe

- Cooperative hierarchical resource management
  - Hardware threads
  - Hierarchy of schedulers
  - Cooperative request/release/allocation of threads up and down the hierarchy

## Lithe Implementation

- Harts: hardware threads, one per physical core/hardware context
- Execution contexts

## Conventional System Vs Lithe

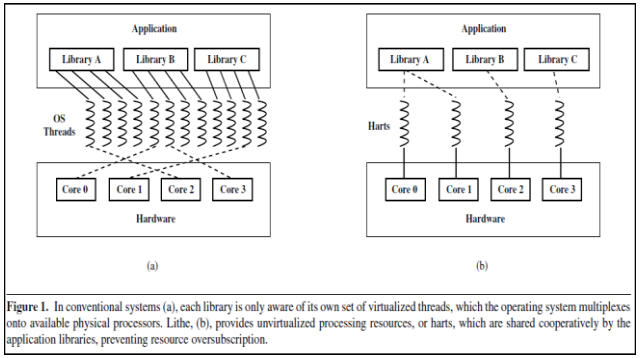


**Figure 1.** In conventional systems (a), each library is only aware of its own set of virtualized threads, which the operating system multiplexes onto available physical processors. Lithe, (b), provides unvirtualized processing resources, or harts, which are shared cooperatively by the application libraries, preventing resource oversubscription.

Image source: Pan et al., PLDI 2010

## Lithe Interface

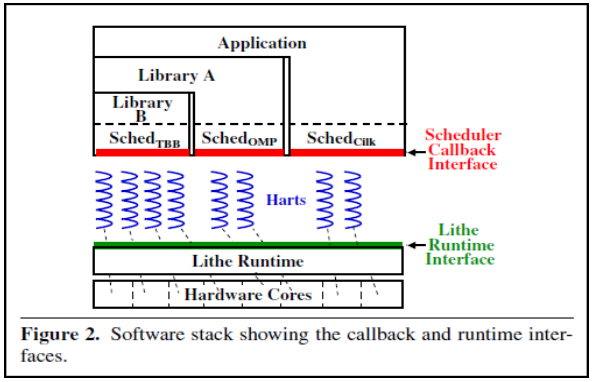


**Figure 2.** Software stack showing the callback and runtime interfaces.

Image source: Pan et al., PLDI 2010

## Lithe Interface

- Scheduler -> implements Lithe callback interface for a library
- Runtime -> invokes the appropriate library's callback

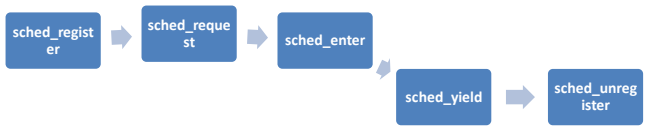


Image creator: Shantonu Hossain

# Example

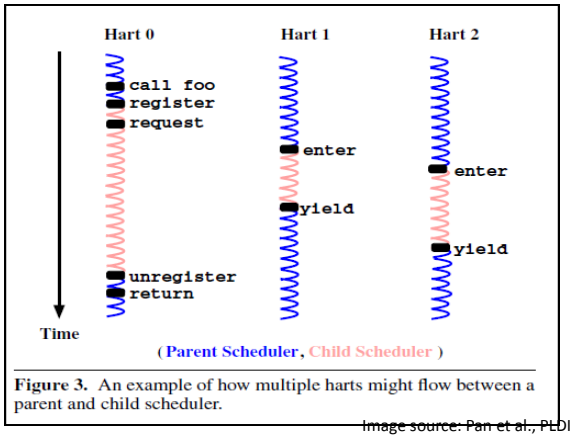


**Figure 3.** An example of how multiple harts might flow between a parent and child scheduler.

Image source: Pan et al., PLDI 2010

# Lithe Runtime Functions and Scheduler Callbacks

**Table 1.** The Lithe runtime functions and their corresponding scheduler callbacks.

| Lithe Runtime Interface | Scheduler Callback Interface |
|---|---|
| `sched_register(sched)` | `register(child)` |
| `sched_unregister()` | `unregister(child)` |
| `sched_request(nharts)` | `request(child, nharts)` |
| `sched_enter(child)` | `enter()` |
| `sched_yield()` | `yield(child)` |
| `sched_reenter()` | `enter()` |
| `ctx_init(ctx, stack)` | N/A |
| `ctx_fini(ctx)` | N/A |
| `ctx_run(ctx, fn)` | N/A |
| `ctx_pause(fn)` | N/A |
| `ctx_resume(ctx)` | N/A |
| `ctx_block(ctx)` | `block(ctx)` |
| `ctx_unblock(ctx)` | `unblock(ctx)` |

Image source: Pan et al., PLDI 2010

# Parallel Quicksort Example

```
1  void sort(vector *v)
2  {
3     par_sort_sched sched;
4     sched.q.init();
5
6     lithe_register(&sched);
7
8     lithe_request(MAX_NUM_HARTS);
9
10    par_sort(v, &sched.q);
11    vector *next;
12    while (sched.q.dequeue(&next))
13       par_sort(next, &sched.q);
14
15    lithe_unregister();
16 }
17
18 void par_sort(vector *v, queue *q)
19 {
20    if (v->length < 1000)
21       seq_sort(v);
22
23    vector *left, *right;
24    v->partition(&left, &right);
25    q->enqueue(right);
26
27    par_sort(left, q);
28 }
29
30 void par_sort_enter(par_sort_sched *sched)
31 {
32    vector *next;
33    while (sched->q.dequeue(&next))
34       par_sort(next, &sched->q);
35    lithe_yield();
36 }
```

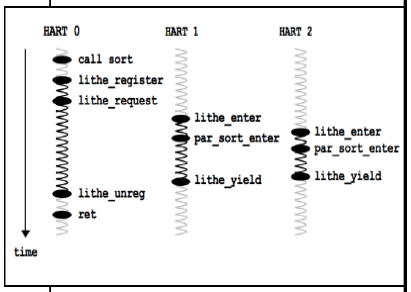


Image source: Pan et al., PLDI 2010

# Lithe – SPMD Scheduler

```
1  void spmd_spawn(int N, void (*func)(void*), void *arg) {
2     SpmdSched *sched = new SpmdSched(N, func, arg);
3     sched_register(sched);
4     sched_request(N-1);
5     sched->compute();
6     sched_unregister();
7     delete sched;
8  }
9
10 void SpmdSched::compute() {
11    while (/* unstarted tasks */)
12       func(arg);
13 }
14
15 void SpmdSched::enter() {
16    if (/* unblocked paused contexts */)
17       ctx_resume(/* next unblocked context */);
18    else if (/* requests from children */)
19       sched_enter(/* next child scheduler */);
20    else if (/* unstarted tasks */)
21       ctx = new SpmdCtx();
22       ctx_run(ctx, start);
23    else sched_yield();
24 }
25
26 void SpmdSched::start() {
27    compute();
28    ctx_pause(cleanup);
29 }
30
31 void SpmdSched::cleanup(ctx) {
32    delete ctx;
33    if (/* not all tasks completed */)
34       sched_reenter();
35    else
36       sched_yield();
37 }
```

Image source: Pan et al., PLDI 2010