Lecture 27:
Under the Hood, Part 1:
Implementing Message Passing

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2016
Today’s Theme
Message passing model (abstraction)

- Threads operate within their own **private address spaces**
- Threads **communicate** by sending/receiving messages
  - **send**: specifies recipient, buffer to be transmitted, and optional message identifier (“tag”)
  - **receive**: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2

Illustration adopted from Culler, Singh, Gupta
**Message passing systems**

- Popular software library: **MPI** (message passing interface)
- Hardware need not implement system-wide loads and stores to execute message passing programs (need only be able to communicate messages)
  - Can connect *commodity systems* together to form large parallel machine
    (message passing is a programming model for **clusters**)

---

**IBM Blue Gene/P Supercomputer**

- **Rack**: 32 node cards
  - **Node Card**: 32 chips, 4x4x2
    - 32 compute, 0-2 IO cards
  - **Compute Card**: 1 chip, 40 DRAMs
  - **Chip**: 4 processors
  - 13.0 GF/s, 8 MB EDRAM
  - 13.6 GF/s, 2 or 4 GB DDR
- **System**: 72 racks
  - **Cabled**: 8x8x16
  - 1 PF/s, Up to 288 TB
  - 14 TF/s, Up to 4 TB
  - 435 GF/s, Up to 128 GB

---

*Image credit: IBM*
Network Transaction

- **One-way transfer** of information from a **source output buffer** to a **destination input buffer**
  - causes some action at the destination
    - e.g., deposit data, state change, reply
  - occurrence is not directly visible at source
Shared Address Space Abstraction

- Fundamentally a two-way request/response protocol
  - writes have an acknowledgement
Key Properties of SAS Abstraction

- **Source and destination addresses are specified by source of the request**
  - a degree of logical coupling and trust
- **No storage logically “outside the application address space(s)”**
  - may employ temporary buffers for transport
- **Operations are fundamentally request-response**
- **Remote operation can be performed on remote memory**
  - logically does not require intervention of the remote processor
Message Passing Implementation Options

**Synchronous:**
- Send completes after matching receive and source data sent
- Receive completes after data transfer complete from matching send

**Asynchronous:**
- Send completes after send buffer may be reused
Synchronous Message Passing

- Data is not transferred until target address is known
- Limits contention and buffering at the destination
- Performance?

1. Initiate send
2. Address translation
3. Local/remote check
4. Send-ready request
5. Remote check for posted receive (assume success)
6. Reply transaction
7. Bulk data transfer
   Source VA —> Dest VA

Send(Pdest, local VA, len)
Receive(Psrc, local VA, len)

Data-transfer request
Send-ready request
Receive-ready reply
Tag check
Wait
Asynchronous Message Passing: Optimistic

- **Good news:**
  - source does not stall waiting for the destination to receive

- **Bad news:**
  - storage is required within the message layer (?)

## Diagram

1. Initiate send
2. Address translation
3. Local/remote check
4. **Send data**
5. Remote check for posted receive; on fail, **allocate data buffer**

![Diagram](attachment:image.png)
Asynchronous Message Passing: Conservative

- Where is the buffering?
- Contention control? Receiver-initiated protocol?
- What about short messages?

1. Initiate send
2. Address translation
3. Local/remote check
4. Send-ready request
5. Remote check for posted receive (assume fail); **record send-ready**
6. Receive-ready request
7. Bulk data reply
   - Source VA —> Dest VA

**Source**
- Send(Pdest, local VA, len)
- Send-ready request
- Resume computing
- Tag match

**Destination**
- Receive(Psrc, local VA, len)
- Receive-ready request
- Data-transfer reply
Key Features of Message Passing Abstraction

- **Source knows send address, destination knows receive address**
  - after handshake they both know both

- **Arbitrary storage “outside the local address spaces”**
  - may post many sends before any receives

- **Fundamentally a 3-phase transaction**
  - includes a request / response
  - can use optimistic 1-phase in limited “safe” cases
    - credit scheme
Challenge: Avoiding **Input Buffer Overflow**

- This requires **flow-control on the sources**

**Approaches:**

1. Reserve space per source (**credit**)  
   - when is it available for reuse? (utilize ack messages?)

2. Refuse input when full  
   - what does this do to the interconnect?  
     - backpressure in a reliable network  
     - tree saturation? deadlock?  
     - what happens to traffic not bound for congested destination?

3. **Drop packets (?)**

4. ???
Challenge: Avoiding **Fetch Deadlock**

- **Must continue accepting messages**, even when cannot source msgs
  - what if incoming transaction is a request?
  - each may generate a response, which cannot be sent!
  - what happens when internal buffering is full?

**Approaches:**

1. **Logically independent request/reply networks**
   - physical networks
   - virtual channels with separate input/output queues

2. **Bound requests and reserve input buffer space**
   - $K(P-1)$ requests + $K$ responses per node
   - service discipline to avoid fetch deadlock?

3. **NACK on input buffer full**
   - NACK delivery?
Implementation Challenges: Big Picture

- **One-way transfer** of information
- **No global knowledge**, nor global control
  - barriers, scans, reduce, global-OR give fuzzy global state
- **Very large number of concurrent transactions**
- **Management of input buffer resources**
  - many sources can issue a request and over-commit destination before any see the effect
- **Latency is large enough that you are tempted to “take risks”**
  - e.g., optimistic protocols; large transfers; dynamic allocation
Lecture 27: Implementing Parallel Runtimes, Part 2

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2016
Objectives

- What are the costs of using parallelism APIs?
- How do the runtimes operate?
Basis of Lecture

- This lecture is based on runtime and source code analysis of Intel’s open source parallel runtimes
  - OpenMP – https://www.openmprtl.org/
  - Cilk – https://bitbucket.org/intelcilkruntime/intel-cilk-runtime

- And using the LLVM compiler
  - OpenMP – part of LLVM as of 3.8
  - Cilk - http://cilkplus.github.io/
OpenMP and Cilk

- What do these have in common?
  - pthreads

- What benefit does abstraction versus implementation provide?
Simple OpenMP Loop Compiled

- What is this code doing?
- What do the OpenMP semantics specify?
- How might you accomplish this?

```c
extern float foo( void );
int main ( int argc, char** argv ) { 
    int i;
    float r = 0.0;
    #pragma omp parallel for schedule(dynamic) reduction(+:r)
    for ( i = 0; i < 10; i ++ ) {
        r += foo();
    }
    return 0;
}
```

Example from OpenMP runtime documentation
**Simple OpenMP Loop Compiled**

```c
extern float foo( void );
int main (int argc, char** argv) {
    static int zero = 0;
    auto int gtid;
    auto float r = 0.0;
    __kmpc_begin( & loc3, 0 );
    gtid = __kmpc_global thread num( & loc3 );
    __kmpc_fork call( &loc7, 1, main_7_parallel_3, &r );
    __kmpc_end( & loc0 );
    return 0;
}
```

Call a function in parallel with the argument(s)
Simple OpenMP Loop Compiled

- **OpenMP “microtask”**
  - Each thread runs the task
- **Initializes local iteration bounds and reduction**
- **Each iteration receives a chunk and operates locally**
- **After finishing all chunks, combine into global reduction**

```c
struct main_10_reduction_t_5 { float r_10_rpr; };

void main_7_parallel_3( int *gtid, int *btid, float *r_7_shp ) {
    auto int i_7_pr;
    auto int lower, upper, liter, incr;
    auto struct main_10_reduction_t_5 reduce;
    reduce.r_10_rpr = 0.F;
    liter = 0;
    __kmpc_dispatch_init_4( &loc7, *gtid, 35, 0, 9, 1, 1 );
    while ( __kmpc_dispatch_next_4( &loc7, *gtid, &liter, &lower, &upper, &incr ) ) {
        for ( i_7_pr = lower; upper >= i_7_pr; i_7_pr ++ )
            reduce.r_10_rpr += foo();
    }
    switch( __kmpc_reduce_nowait( &loc10, *gtid, 1, 4, &reduce, main_10_reduce_5, &lck ) ) {
    case 1:
        *r_7_shp += reduce.r_10_rpr;
        __kmpc_end_reduce_nowait( &loc10, *gtid, &lck);
        break;
    case 2:
        __kmpc_atomic_float4_add( &loc10, *gtid, r_7_shp, reduce.r_10_rpr );
        break;
    default:;
    }
}
```

Example from OpenMP runtime documentation
Simple OpenMP Loop Compiled

- All code combined

```c
extern float foo( void );
int main (int argc, char** argv) {
    static int zero = 0;
    auto int gtid;
    auto float r = 0.0;
    __kmpc_begin( & loc3, 0 );
    gtid = __kmpc_global_thread_num( & loc3 );
    __kmpc_fork_call( &loc7, 1, main_7_parallel_3, &r );
    __kmpc_end( & loc0 );
    return 0;
}

struct main_10_reduction_t_5 { float r_10_rpr; }
static kmp_critical_name lck = { 0 };
static ident_t loc10;

void main_10_reduce_5( struct main_10_reduction_t_5 *reduce_lhs, struct main_10_reduction_t_5 *reduce_rhs ) {
    reduce_lhs->r_10_rpr += reduce_rhs->r_10_rpr;
}

void main_7_parallel_3( int *gtid, int *btid, float *r_7_shp ) {
    auto int i_7_pr;
    auto int lower, upper, liter, incr;
    auto struct main_10_reduction_t_5 reduce;
    reduce.r_10_rpr = 0.F;
    liter = 0;
    __kmpc_dispatch_init_4( & loc7,*gtid, 35, 0, 9, 1, 1 );
    while ( __kmpc_dispatch_next_4( & loc7, *gtid, &liter, &lower, &upper, &incr ) ) {
        for( i_7_pr = lower; upper >= i_7_pr; i_7_pr ++ )
            reduce.r_10_rpr += foo();
    }
    switch( __kmpc_reduce_nowait( & loc10, *gtid, 1, 4, &reduce, main_10_reduce_5, &lck ) ) {
    case 1:
        *r_7_shp += reduce.r_10_rpr;
        __kmpc_end_reduce_nowait( & loc10, *gtid, &lck);
        break;
    case 2:
        __kmpc_atomic_float4_add( & loc10, *gtid, r_7_shp, reduce.r_10_rpr );
        break;
    default:
        break;
    }
}
```

Example from OpenMP runtime documentation
Fork Call

- “Forks” execution and calls a specified routine (microtask)
- Determine how many threads to allocate to the parallel region
- Setup task structures
- Release allocated threads from their idle loop
Iteration Mechanisms

- Static, compile time iterations
  - \texttt{kmp\_for\_static\_init}
  - Compute one set of iteration bounds

- Everything else
  - \texttt{kmp\_dispatch\_next}
  - Compute the next set of iteration bounds
OMP Barriers

- Two phase -> gather and release
  - Gather non-master threads pass, master waits
  - Release is opposite

- Barrier can be:
  - Linear
  - Tree
  - Hypercube
  - Hierarchical
OMP Atomic

- Can the compiler do this in a read-modify-write (RMW) op?
- Otherwise, create a compare-and-swap loop

```c
T* val;
T update;
#pragma omp atomic
    *val += update;
```

If $T$ is int, this is “lock add ...”.
If $T$ is float, this is “lock cmpxchg ...”

Why?
OMP Tasks

- #pragma omp task depend (inout:x) ...

- Create microtasks for each task
  - Track dependencies by a list of address / length tuples
Cilk

- Covered in Lecture 5
- We discussed the what and why, now the how
Simple Cilk Program Compiled

- What is this code doing?
- What do the Cilk semantics specify?
- Which is the child? Which is the continuation?

```c
int fib(int n) {
    if (n < 2)
        return n;
    int a = cilk_spawn fib(n-1);
    int b = fib(n-2);
    cilk_sync;
    return a + b;
}
```
How to create a continuation?

- Continuation needs all of the state to continue
  - Register values, stack, etc.

- What function allows code to jump to a prior point of execution?
  - `Setjmp(jmp_buf env)`
    - Save stack context
    - Return via `longjmp(env, val)`
    - `Setjmp` returns 0 if saving, val if returning via `longjmp`
Basic Block

- **Unit of Code Analysis**

- **Sequence of instructions**
  - Execution can only enter at the first instruction
    - Cannot jump into the middle
  - Execution can only exit at the last instruction
    - Branch or Function Call
    - Or the start of another basic block (fall through)
Simple Cilk Program Revisited

entry

setjmp

!0

fib(n-2)

fib(n-1)

setjmp

Is sync?

!0

f1 + f2

f1 + f2

ret

Leave frame

Save Continuation

Cilk RTS sync

maybe

parallel

serial

0

0

0

0

!0

0
Cilk Workers

- While there may be work
  - Try to get the next item from our queue
  - Else try to get work from a random queue
  - If there is no work found, wait on semaphore

- If work item is found
  - Resume with the continuation’s stack
Thread Local Storage

- Linux supports thread local storage
  - **New: C11 - __Thread_local** keyword
    - one global instance of the variable per thread
  - Compiler places values into .tbss
  - OS provides each thread with this space

- Since Cilk and OpenMP are using pthreads
  - These values are in the layer below them
DEMO