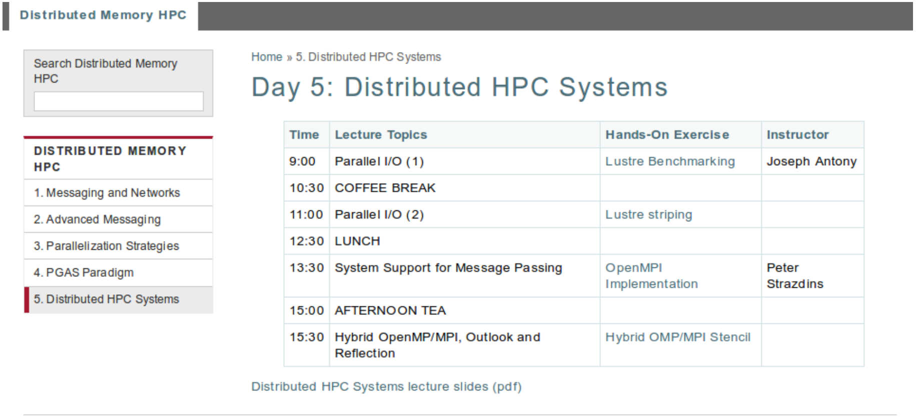
Distributed HPC Systems
ASD Distributed Memory HPC Workshop

Computer Systems Group

Research School of Computer Science
Australian National University
Canberra, Australia

November 03, 2017

Day 5 – Schedule



Distributed Memory HPC

Search Distributed Memory HPC

DISTRIBUTED MEMORY HPC

1. Messaging and Networks
2. Advanced Messaging
3. Parallelization Strategies
4. PGAS Paradigm
5. Distributed HPC Systems

Home » 5. Distributed HPC Systems

Day 5: Distributed HPC Systems

Time	Lecture Topics	Hands-On Exercise	Instructor
9:00	Parallel I/O (1)	Lustre Benchmarking	Joseph Antony
10:30	COFFEE BREAK		
11:00	Parallel I/O (2)	Lustre striping	
12:30	LUNCH		
13:30	System Support for Message Passing	OpenMPI Implementation	Peter Strazdins
15:00	AFTERNOON TEA		
15:30	Hybrid OpenMP/MPI, Outlook and Reflection	Hybrid OMP/MPI Stencil	

Distributed HPC Systems lecture slides (pdf)

Outline

Hands-on Exercise: Lustre Benchmarking

Outline

Hands-on Exercise: Lustre Striping

Outline

Operating System Support

- distributed memory supercomputer nodes have many cores, typically in a NUMA configuration
- OS must support efficient (remote) process creation
  - typically the TCP transport will be used for this

  The MPI runtime must also use efficient `ssh` 'broadcast' mechanism
  - e.g. on Vayu (Raijin's predecessor), a 1024 core job required 2s for pre-launch setup, 4s to launch processes
- the OS must avoid *jitter*, particularly problematic for large-scale synchronous computations
  - support **process affinity**: binding processes/threads to particular cores (e.g. Linux `get/set_cpu_affinity()`)
  - support **NUMA affinity**: ensure (by default) memory allocations is on the adjacent NUMA domain to the core
  - support efficient interrupt handling (from network traffic)
  - otherwise ensure all system calls are handled quickly and evenly (limit amount of 'book-keeping' done in any kernel mode switch)

  Alternately devote 1 core to OS to avoid this (IBM Blue Gene)

Interrupt Handling

- by default, all cores handle incoming interrupts equally (**SMP**)
- potentially, interrupts cause high (L1) cache and TLB pollution, as well as delay (switch to kernel context, time to service) threads running on the servicing core
- solutions:
  - OS can consider handling all on one core (which has no compute-bound threads allocated to it)
  - two-level interrupt handling (used on GigE systems):
    - **top-half interrupt handler** simply saves any associated data and initiates the **bottom-half handler**
      e.g. (for a network device) handler simply deposits incoming packets into an appropriate queue
    - the core running the interrupt's destination process should service the **bottom-half interrupt**
  - use OS bypass mechanisms (e.g. Infiniband): initiate RDMA transfers from user-level, detect incoming transfers instead by polling
    - an interrupt informs initiating process when transfer complete
    - also enables very fast latencies! ($< 1\mu s$)

MPI Profiling Support

Australian National University

- how is it that we can turn on MPI profilers without even having to recompile our programs? (`module load ipm; mpirun -np 8 ./heat`)
- in MPI's profiling layer **PMPI**, every MPI function (e.g. `MPI_Send()`) by default 'points' to a matching PMPI function (e.g. `PMPI_Send()`):

```
#pragma weak MPI_Send = PMPI_Send
int PMPI_Send(void *buf, ... ) {
  /*do the actual Send operation*/ .... }
```

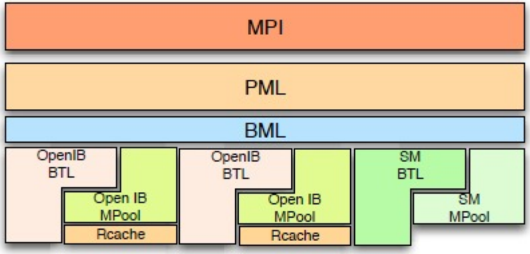
- thus the app. or a library (e.g. IPM) can provide a customized version of the function (i.e. for profiling), e.g.

```
static int nCallsSend = 0;
int MPI_Send(void *buf, ...) {
  nCallsSend++; PMPI_Send(buf, ...); }
```

- MPI provides a `MPI_Pcontrol(int level, ...)` function which by default is a no-op but may be similarly redefined
  - IPM provides `MPI_Pcontrol(int level, char *label)`
  - `level` $= +1$ (-1): start (end) profiling a region, called `label`
  - `level` $= 0$: invoke a custom event, called `label`

OpenMPI Architecture

- based on the **Modular Component Architecture** (**MCA**)
- each component framework within the **MCA** is dedicated to a single task, e.g. providing parallel job control or performing collective operations
- upon demand, a framework will discover, load, use, and unload components
- OpenMPI component schematic:



(courtesy L. Graham et al, *Open MPI: A Flexible High Performance MPI*, EuroPVMMPI'06)

OpenMPI Components

- **MPI**: handles top-level MPI function calls
- **Collective Communications**: the back-end of MPI collective operations has SM-optimizations
- **Point-to-point Management Layer** (PML): manages all message delivery (including MPI semantics). Control messages are also implemented in the PML
  - handles message matching, fragmentation and re-assembly,
  - selects protocols depending on message size and network capabilities
  - for non-blocking sends and receives, a callback function is registered, to be called when a matching transfer is initiated
- **BTL Management Layer** (BML): during `MPI_Init()`, discovers all available BTL components, and which processes each of them will connect to
  - users can restrict this, i.e. `mpirun --mca btl self,sm,tcp -np 16 ./mpi_program`

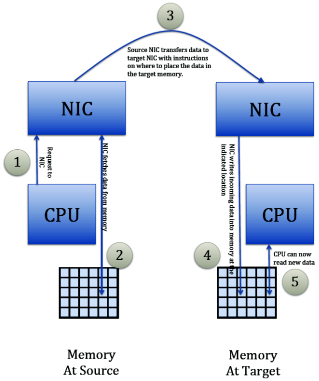
OpenMPI Components (II)

- **Byte-Transfer-Layer Layer** (BTL): handles point-to-point data delivery
  - the default shared memory BTL copies the data twice: from the send buffer to a shared memory buffer, then to the receive buffer
  - connections between process pairs are lazily set up when the first message is attempted to be sent
- **MPool** (memory pool): provides send/receive buffer allocation & registration services
  - registration is required on IB & similar BTLs to 'pin' memory; this is costly and cannot be done as a message arrives
- **RCache** (registration cache): allows buffer registrations to be cached for later messages

Note: whenever an MPI function is called, the implementation may choose to search all message queues of the active BTLs for recently arrived messages (this enables system-wide 'progress').

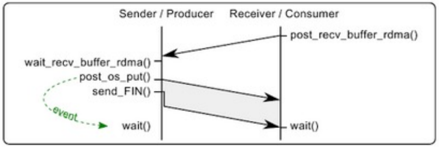
Message Passing Protocols via RDMA

- message passing protocols are usually implemented in terms of **Remote Direct Memory Access** (**RDMA**) operations
- each process contains *queues*: a pre-defined location in memory to buffer send or receive requests
    - these requests specify the message 'envelope' (source/destination process id, tag, size)
    - remote processes can write to these queues
      also can read/write into buffers (once they know its address)

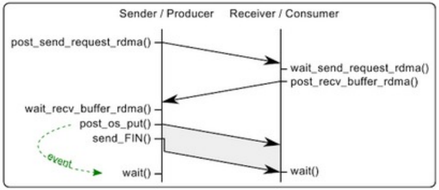

3
Source NIC transfers data to target NIC with instructions on where to place the data in the target memory.
NIC
NIC
1
Request to NIC
NIC fetches data from memory
NIC writes incoming data into memory at the indicated location
CPU
CPU
2
4
CPU can now read new data
5
Memory At Source
Memory At Target

(courtesy Grant & Olivier, *Networks and MPI Cluster Computing*)

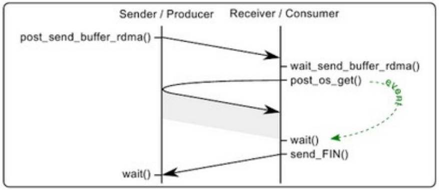
Message Passing Protocols via RDMA



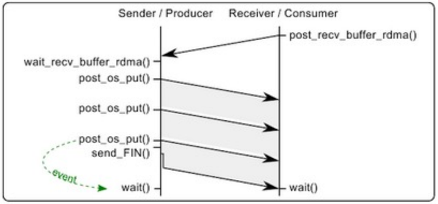
(a) Consumer Initiated RDMA write protocol



(b) Producer Initiated RDMA write protocol



(c) Producer Initiated RDMA read protocol



(d) Advanced RDMA write based protocol

(courtesy Danalis et at, *Gravel: A Communication Library to Fast Path MPI*, EuroMPI'08

Consumer-initiated RDMA-write Protocol

This supports the usual rendezvous protocol.

- consumer sends the receive message envelope (with the buffer address) to producer's *receive-info queue*
- when producer posts a matching send, its reads this message envelope (or blocks till it arrives)
- producer transfers data via an RDMA-write, then sends the send message envelope to consumer's *RDMA-fin queue*
- the consumer blocks till this arrives

The **Producer-initiated RDMA-write Protocol** supports `MPI_Recv(..., MPI_ANY_SOURCE)`):

- producer sends the send message envelope to the consumer's *send-info queue*.
- when consumer posts a matching receive, it reads this envelope from the queue (or blocks until one arrives). Then, it continues as above.

Other RDMA Protocols

The **Producer-initiated RDMA-read Protocol** can also support the rendezvous protocol:

- the producer sends the message envelope (with send buffer address) to the consumer's *send-info queue*
- when the consumer posts a matching receive, it reads the envelope from the ledger (or blocks till it arrives)
- it then does an RDMA-read to perform the transfer
- when complete, it sends a the message envelope to the producer's *rdma-fin queue*

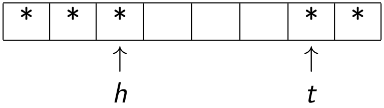
Eager protocol: producer writes the data into a pre-defined remote buffer and then sends the message envelope to consumer's *send-info queue*.

RDMA Queue Implementation

Generally, the producer (remote node) adds items to the queues, the consumer (local node) removes them. Issues:

- how does producer know the addresses of remote queues/buffers?
- are per-connection queues and buffers needed?
- what happens if the producer gets too far ahead?

Implementation is generally done via a **ring buffer** with fixed size entries:



Adding an element involves the remote:

- fetching of h and t (check $h < t$)
- increment of h
- writing the new entry at the hth element,

adding to the latency! A similar scheme can be used for the data buffers.

Case Study in System-related Performance Issues

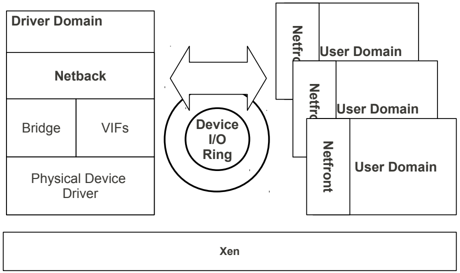
Profiling the MetUM global atmosphere model on the Vayu IB cluster, Jan 2012 (p2-4,7,14-16,18,9)

- without process and NUMA affinity, there is vastly greater variability in performance
- loss of NUMA affinity even on 2 processes (out of 1024) resulted in 30% loss of performance
- an algorithm requiring many IB connections per process created very large startup costs (and was from then much slower!)
  - involves the creation of many buffers for queues etc, their registration and exchange to the remote process
  - avoid such algorithms where possible!
- for large number of process, required increasing amounts of pinned memory (even though application data / process is decreasing!)

Message Passing Support on Virtualized Clusters

Virtualized HPC nodes (e.g. on AWS) have several advantages:

- users can fully customize their environment, better security
- OS is no longer tied to physical nodes (flexible Windows/Linux systems)



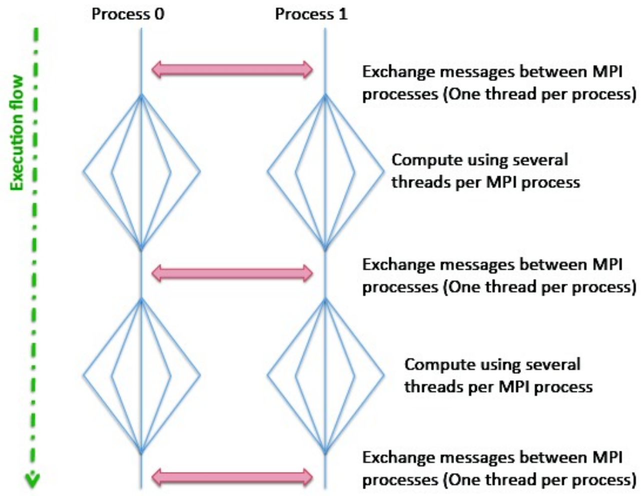
However, virtualized (network) I/O inherently has a number of overheads; also, they usually use TCP/IP transports (e.g. 10GigE). Solutions include:

- allowing the ‘user’ OS to directly access network interfaces e.g. VMM-bypass (Xen) or SR-IOV (currently works on KVM and IB) (SR-IOV allows a network adaptor to be shared by multiple user OSs)
- TCP/IP protocol processing offload, to specialized NICs, or to a dedicated core on the node (in the case of Xen, running the Driver Domain)

Hands-on Exercise: OpenMPI Implementation

Outline

Hybrid OpenMP / MPI Parallelism: Ideas


Process 0
Process 1
Execution flow
Exchange messages between MPI processes (One thread per process)
Compute using several threads per MPI process
Exchange messages between MPI processes (One thread per process)
Compute using several threads per MPI process
Exchange messages between MPI processes (One thread per process)

(courtesy Grant & Olivier, *Networks and MPI for Cluster Computing*)

Hybrid OpenMP / MPI Parallelism: Motivations

- message passing and shared memory programming paradigms are not mutually exclusive
  - we can (easily) create and use OpenMP threads within an MPI application
- almost all supercomputers today have large (8+ core) nodes connected to a high speed network
  - i.e. native shared / distributed memory hardware within / between nodes
- natural to reflect this in the programming model
- idea: use OpenMP to parallelize an application over the cores (or NUMA domains) within a node, and MPI to parallelize across nodes
  - a hierarchical programming model better reflects the increasing complexity of nodes (core count, NUMA domains) and should have performance advantages

Hybrid OpenMP/MPI: Possible Advantages

Australian National University

- reduces the number of MPI processes and associated overheads (creation, connection management, memory footprint)
  - also reduce communication startups and (sometimes) volume
- collectives are (should be) faster via native shared memory
- a dedicated thread for MPI can improve messaging performance (overlap communication with computation)
- balance dynamically varying loads between processes (on one node)
  - OpenMP is capable of handling threads dynamically, in a relatively lightweight fashion
- benefits of data sharing between threads: enhanced shared cache performance

  (however, pure MPI will minimize cache coherency overheads
- obtain extra parallelization when the MPI implementation restricts the number of processes (e.g. NAS BT benchmark restricted to $p = k^2$ processes)

MPI Threading: Vector Mode

Outside parallel regions, the master thread calls MPI.
e.g. Jacobi `heat.c` program

```
do {
  iter++;
  jst = rank*chk+1;
  jfin = (jst+chk > Ny-1)? Ny-1: jst+chk;
  #pragma omp parallel for private(i)
  for (j = jst; j < jfin; j++)
  for (i = 1; i < Nx-1; i++) {
    tnew[j*Nx+i] = 0.25*(told[j*Nx+i+1]+...+told[(j-1)*Nx+i]);
  // end of parallel region - implicit barrier

  if (rank+1 < size) {
    jst = rank*chk+chk;
    MPI_Send(&tnew[jst*Nx],Nx, MPI_DOUBLE, rank+1, 2, ...);
  }
  ...
} while (iter < Max_iter);
```

Relatively easy incremental parallelization using OpenMP directives.

MPI Threading: Thread Mode

A single thread handles MPI while others run. Here `heat.c` becomes

```
#pragma omp parallel private(tid, iter, j, i, jst, jfin)
{ int tid = omp_get_thread_num(),
      nthr = omp_get_num_threads()-1, chkt;
do { iter++;
  if (tid > 0) { //do the computation
    jst = rank*chk+1;
    jfin = (jst+chk > Ny-1)? Ny-1: jst+chk;
    chkt = (jfin - jst + nthr - 1) / nthr;
    jst += chkt*tid; jfin = (jst+chkt > jfin)? jfin: jst+chkt;
    for (j = jst; j < jfin; j++)
      ...
  } else { //thread 0 handles MPI
    if (rank+1 < size) { //race hazard here?
      jst = rank*chk+chk;
      MPI_Send(&tnew[jst*Nx], Nx, MPI_DOUBLE, rank+1, 2, ...);
    } ...
  } ...
} while (iter < Max_iter); } //parallel region
```

Sychronization of thread 0 and others problematic; blows up code; non-incremental.

MPI Thread Support: The 4 Levels

Australian National University

- `MPI_THREAD_SINGLE`: only one thread will execute (standard MPI-only application)
- `MPI_THREAD_FUNNELED`: only the thread that initialized MPI may call MPI (usually the master thread).
  In **thread mode**, inside a parallel region, we would need

```
#pragma omp master      // surround with barriers if a
  MPI_Send(data, ...); // race hazard on data is possible
```

- `MPI_THREAD_SERIALIZED`: only one thread will may call at any time.
  In **thread mode**, inside a parallel region, we would need:

```
#pragma omp barrier
#pragma omp single
  MPI_Send(data, ...);
#pragma omp barrier
```

- `MPI_THREAD_MULTIPLE` any threads may call MPI at any time MPI library has to ensure **thread safety** - may have high overhead!

Mapping of Threads and Processes

- generally, per node, #threads x #processes per = #CPUs
  - possibly #virtual CPUs, if hyperthreading is available
- consider an 8-core 2-socket node
  -
p_0							
t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7

    (one process per node)

    May get excessive synchronization overheads and NUMA penalties; 1 thread may not be enough to saturate network
  -
p_0				p_1			
t_0	t_1	t_2	t_3	t_0	t_1	t_2	t_3

    (one process per socket)

    Once processes are pinned to sockets, optimizes NUMA accesses. May be a 'sweet spot': low synchronization overhead, good L3 cache re-use between threads, reduced number of processes.
  -
p_0		p_1		p_2		p_3	
t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1

    (two processes per socket)

    Possibly reduced benefits. May be suitable for dynamic thread parallelism (1-4 threads per process).

Hybrid OpenMP / MPI Job Launch

- in application, must change `MPI_Init(&argc, &argv)` with:
  `MPI_Init_thread(&argc, &argv, required, &provided)`
  where `int required` is one of the 4 MPI levels of thread support (and `provided` is set to what your MPI implementation will give you!)
- in your batch file
  - specify the total number of cores for the batch system (as before)
  - specify the number of thread per process,
    e.g. `export OMP_NUM_THREADS = 4`
  - specify the number of processes per node (or socket) for `mpirun`,
    e.g. `mpirun -np 64 -npernode 8 ...`

When to Try Hybrid OpenMP / MPI?

Australian National University

- when the scalability of your pure MPI application is lower than desired
  - or when the L3 cache performance is low due to capacity-caused misses
- when MPI parallelization is only partial (e.g. 2D on a 3D problem) (or otherwise limited)
  - using OpenMP to parallelize 3rd dimension may leads to better data 'shape' per CPU
- when problem size is limited by memory per process (important in 'high-end' supercomputing)
- when the potentially large extra effort of refactoring and maintaining the hybrid code is worth it! (especially if you want to use thread mode!)

Overview: Outlook and Review

- the shared memory coherency wall
- multicore/manycore processors
- 'high end' systems
- distributed memory programming models

The Coherency Wall: Cache Coherency Considered Harmful!

Recall that hardware shared memory requires a network connecting caches to main memory with a **coherency protocol** for correctness.

- standard protocols requires a broadcast message for each invalidation
  - standard MOESI protocol also requires a broadcast on every miss
  - energy cost of each is $O(p)$; overall cost is $O(p^2)$!
  - also causes contention (& delay) in the network (worse than $O(p^2)$?)
- **directory-based** protocols better, but only for lightly-shared data
  - for each cached line, need a bit vector of length p: $O(p^2)$ storage cost
- **false sharing** in any case results in wasted traffic
- **atomic instructions** (essential for locks etc) sync the memory system down to the LLC, cost $O(p)$ energy each!
- cache line size is sub-optimal for messages on on-chip networks

Multicore/Manycore Processor Outlook

- diversity in approaches; post-RISC ideas will still be tried
  - "two strong oxen or 1024 chickens" (Seymour Cray, late 80's) debate to continue
- energy issues will generally increase in prominence
- overcoming the **memory wall** continues to be a major factor in design
  - increasing portion of design effort and chip area devoted to data movement
- predict the **coherency wall** will begin to bite at 32 cores
  - long-term future for inter-socket coherency?
- are we now at The End of Moores Law?
  Or will Extreme Ultraviolet Lithography (EUV) allow feature size to shrink from 20nm $\rightarrow$ 10 nm $\rightarrow$ 7nm?
- domain-specific approaches will become more prevalent
  e.g. the emerging killer HPC app: **deep learning**
  - Google's TPU: a 256 $\times$ 256 **systolic array** for 8-bit matrix multiply for AI applications

Outlook – High End (Massively Parallel) Systems

Australian National University

- the (US) Path to Exascale (2020–2025)
  - (compute) parallelism a thousand-fold greater than todays systems
  - memory and I/O performance to improve accordingly with increased computational rates and data movement requirements.
  - reliability that enables recovery from faults (probability of hard or soft failures increase with application/system size and running time)
  - energy efficiencies $> 20\times$ today's capabilities
- further ahead, alternative / extreme parallel computing paradigms may emerge:
  - molecular computing (including DNA computing): long times for individual simulations (hours), but size (p) is no problem!
  - quantum computing: search exponential (2^n) spaces in constant time using n **qubits**

Outlook: Distributed Memory Prog. Models

Australian National University

- domain-specific languages offer abstraction over underlying parallel system
  - e.g. the Physis stencil framework
  - a declarative, portable, global-view DSL targeting C/Cuda(+MPI)
  - can apply parallization and various GPU-specific optimizations automatically
  - in future, may be able to apply MPI optimizations also
- will a programming language/model deliver the *silver bullet*? (or even cover devices & cores seamlessly?)
- for large-scale systems, scalability, reliability and tolerance to performance variability are the key concerns
  - **PGAS** and **task-DAG** programming models can deal with distributed memory, both within and across (network-connected) chips
  - may need hierarchical notions of locality (**places**)
  - both can deal with 2nd & 3rd issues

Review of the Message Passing Paradigm

- has **synchronous**, **blocking** and **non-blocking** semantics; what is the difference?
- distribution schemes are basically fixed (need to find start offsets and length of the local portion of the data, using the process id)
- messages can also be used for synchronization
- message passing programs *can* run within a shared memory domain (node or socket); how (e.g. on Raijin)?
  Possible advantages:
  - better separation of the hardware-shared memory (e.g. NUMA) – can be faster
  - **cache coherency** no longer required!
- should this be the default programming paradigm? (e.g. Intel SCC)
  - Kumar et al, The Case For Message Passing On Many-Core Chips:
    or, *the shared memory programming model considered difficult*
    - timing-related issues more prevalent: e.g. data-races, especially with **relaxed memory consistency**
    - no safety / composability / modularity

Review of the Message Passing Paradigm (II)

- for large-scale systems, distributed memory hardware is still essential
  - the network topology and routing strategies have a large impact on performance
  - some notion of locality is needed for acceptable performance
  - system level support is non-trivial, with high memory overheads for message buffers
  - size of system itself may require fault-tolerance to be considered
- message-passing is a highly ubiquitous parallel programming paradigm
  - it can be made efficient, in the best case, with reasonable programming effort
  - in the worst case, dynamically varying and irregular date structures (e.g. Barnes-Hut oct-trees) can be very difficult!
  - we must *explicitly* understand communication patterns and know collective algorithms
  - we have a highly sophisticated middleware (MPI) to support it
  - has well-defined strategies which support large classes of problems
  - it can be combined with the shared memory paradigm with relative ease (reflecting the hierarchic hardware organization of large-scale systems)

Summary

Topics covered today:

- parallel I/O in Lustre filesystems
- system support for message passing (OpenMPI case study)
- hybrid OpenMP / MPI parallelism
- outlook for large scale message passing systems and paradigm
- review

Hands-on Exercise: Hybrid OMP/MPI Stencil