Message passing, middleware and shared memory on clusters

Mats Brorsson
KTH School of Information and Communication Technology

Outline

• Important concepts from last lecture
• Message-passing programming
• MPI — The Message Passing Interface
• Collective communication
• Middleware for MPI and OpenMP
• Software Distributed Shared Memory

Important concepts from last lectures

• A memory consistency model specifies how memory events initiated by one process should be observed by the other processes
• The memory consistency model dictates how the architecture can order reads and writes to the shared memory with respect to each other
• A cache coherence mechanism can help in realising a particular consistency model
• Synchronisation primitives are key to implement shared memory algorithms
  – Atomic instructions, or
  – Load-linked, store-conditional
• High-level synchronisation operations are built upon simple primitives

More Important Concepts

• Shared memory programming models are most often based on threads
• Pthreads, Java threads
• OpenMP (for loops)
• New models are task-centric (lecture 29 Sep)
  – OpenMP tasks (multi vendors)
  – Cilk++ (Intel)
  – Intel TBB (Intel)
  – Microsoft TPL (Microsoft)
  – Wool (SICS)
  – Grand Central Dispatch (Apple)
Machine models—Review

- Distributed memory
  - No physically shared memory
- Scalable interconnection network
- Physically shared memory
- Cache coherence
- Often bus-based interconnect

Parallel programming

- Dominant programming models
  - Shared memory
  - Message-passing
- The parallel programming model is different from the machine model
  - A message-passing model can be implemented on distributed memory machines as well as on shared memory machines
  - A shared memory programming model is best suited for shared memory machines but can be implemented also on distributed memory machines

The message-passing model

- Abstract computational model
- Each process has its own exclusive address space
- Each process runs a copy of the same program in parallel (SPMD — Single program multiple data)
- Process rank and the number of processes are used to divide the work

Message-passing?

- Usually a library routine approach
- Similar routines to determine process identity and number of processes
  - No shared memory!
- Two additional routines (at minimum):
  - send(data, destination, size, tag)
  - receive(buffer, source, size, tag)
MPI

Message-Passing Interface
• Developed by a consortium of parallel computer vendors
• Static processing model – all processes are created when the program is loaded
• The process group MPI_COMM_WORLD is the set of all processes
• http://www.mpi-forum.org/
• Most popular free implementation: MPICH

A minimal set of MPI

• MPI_Init(argc, argv) – Initialisation of MPI
• MPI_Comm_size(MPI_COMM_WORLD, &group_size) – returns the number of processes in variable group_size
• MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) – returns the own process identity
• MPI_Send – Send a message
• MPI_Recv – Receive a message
• MPI_Finalize() – Clean up MPI stuff

MPI Messages

• A message contains a number of elements of some particular datatype
• MPI datatypes
  – Basic types
  – Derived types
• Derived types can be built from basic types
• C types are different from Fortran types

MPI basic datatypes – C

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_INT</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>long</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Sending a message

MPI_Send(&N, 1, MPI_INT, i, tag, MPI_COMM_WORLD);

- &N – address to send buffer
- 1 – the number of elements to send
- MPI_INT – the datatype of message
- i – the receiving process
- tag – used to label this message
- MPI_COMM_WORLD – Communicator

- Buffered and non-blocking

Receiving a message

MPI_Recv(&tmp, 1, MPI_INT, i, tag, MPI_COMM_WORLD, &status);

- &tmp – address to receive buffer
- 1 – the maximum number of elements to receive
- MPI_INT – the datatype of message
- i – the sending process
- tag – used to label this message. This must match the corresponding tag in MPI_Send
- status – information about the message received

- Blocking

Wildcards

- Tags and specification of the sending process makes it easier to write correct code
- Sometimes its better to use wildcards:
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

Receive status

- The status parameter can be used to find out what actually was received
- Some examples:
  - status.MPI_SOURCE
  - status.MPI_TAG
  - MPI_Get_Count(&status, MPI_INT, &count)
Interaction modes
Three common interaction modes:
- Synchronous message-passing
  - A send waits until the receive has been encountered and vice-versa
- Blocking Send/Receive
  - A Send waits until the data has been sent
    - It does not mean it has been received
  - A receive waits until there is data to receive
- NonBlocking Send/Receive
  - The send or receive may return before data has been sent or received
  - Programmer has to explicitly check for completion

Overlapping communication and computation

• NonBlocking Send/Receive
  Process P
  \[ M = 10; \]
  \[ \text{send} \ M \ \text{to} \ Q; \]
  \[ \text{do some useful work} \]
  \[ \text{wait for} \ M \ \text{to be sent} \]
  \[ M = 20; \]
  \[ \text{receive} \ S \ \text{from} \ P; \]
  \[ \text{do some useful work} \]
  \[ \text{wait for} \ S \ \text{to be received} \]
  \[ X = S + 1; \]

Comparison
- Synchronous message-passing is easy to understand but wastes time
- Blocking Send/Receive is also easy to understand and is the normal interaction mode
- NonBlocking Send/Receive can be used to overlap communication and computation

Interaction patterns
- A message-passing program normally has static communication patterns known at compile time
- Communication patterns:
  - One-to-One (Normal send/recv)
  - One-to-Many (e.g. broadcast)
  - Many-to-One (e.g. reductions)
  - Many-to-Many (e.g. circular shift)
One-to-One
• Example: P1 sends 1 to P3

One-to-Many
• Example: P1 broadcasts 1 to all

One-to-Many
• Example: P1 scatters its data to P2 and P3

Many-to-One
• Example: P1 gathers data from P2 and P3
Many-to-One

- Example: A **reduction** of data in P1, P2 and P3 to P1. For example a sum.

Many-to-Many

- Example: A **circular shift** of data

Collective Communication Routines

- **MPI_Barrier()**
- **MPI_Bcast()**
- **MPI_Scatter()**
- **MPI_Gather()**
- **MPI_Alltoall()**
- **MPI_Allgather()**
- **MPI_Allreduce()**
- **MPI_Reduce()**
- **MPI_Reduce_scatter()**
- **MPI_Scan()**

MPI and OpenMP Middleware

- Both MPI and OpenMP programming models need library support to work well
- MPI is to high-level to be supported directly by hardware
- OpenMP is more efficiently implemented if the compiler can generate code for a high-level run-time system
  - As opposed to the operating system
Realizing synchronous MP

Synchronisation messages are small of fixed size
Data messages can be large and of variable size

Asynchronous MP – Naïve impl.

- What are the problems with this?

Asynchronous MP– Robust algorithm

- The three-phase protocol eliminates the need for expensive buffer storage at the receiver
- The simpler protocol can still be used for short messages

An Example OpenMP Implementation
The OpenMP Translator
The OpenMP translator deals with three things:
- Transformation of OpenMP constructs
  - Parses constructs
  - Performs some semantic and syntactic checks
  - Instruments the code with calls to the run-time library
- Handling of data clauses
  - Parses data clauses
  - Performs checks
  - Possibly alter variable declarations
- Instrumentation of OpenMP constructs
  - Interface to performance monitoring tools

The Run-Time Library
The run-time library deals with:
- Thread creation
- Thread synchronization
  - Locks
  - Barriers
- Work-sharing
  - Distribution of loop iterations among threads
- Memory consistency
  - Flush operations

The parallel OpenMP construct
```
#pragma omp parallel
{   foo(omp_get_thread_num()); }
```
```
in_tone_c_pf0( ... ) {
   foo(omp_get_thread_num());
}
in_tone_spawnparallel(in_tone_c_pf0, ... );
```
- The parallel construct forces threads to be created
  - The parallel region is executed in parallel
  - One level of parallelism is supported

What about shared variables?
```
int s, p1;
#pragma omp parallel private(p1)
{   float p2;
    foo(omp_get_thread_num(), &p1);
}
```
- Variables with global scope are normally shared by all threads
- Private variables with a global scope are allocated on each thread's stack during the parallel region and references are modified by the compiler
- Stack allocated variables that are shared are accessed through pointer references
- Stack allocated variables that are private are accessed through the stack pointer
Work-sharing constructs—the for-loop

- The run-time library’s work-sharing primitives directly support for-loops
- The for-loop is translated into:
  - A call to the run-time that initializes the for loop
  - A while-loop that requests iterations until there are no left and does the work
  - A call to the run-time ending the for-loop

```
#pragma omp for schedule(dynamic, 2) lastprivate(lp1)
for (i = 0; i < MAX; i = i + 3) {
    /* Body of the parallel loop */
}
```

The **single** construct

- The singles construct is treated as a for-loop with a single (1) iteration
- The nowait clause causes the compiler to not emit the code for the otherwise implicit barrier

```
#pragma omp single nowait
{
    foo();
}
#pragma omp single
{
    bar();
}
```

The **section** construct

- Each section is treated as an iteration and the sections construct is transformed to a for-loop

```
#pragma omp sections
{
    #pragma omp section
    { A(); }
    #pragma omp section
    { B(); }
}
```

The **critical** construct

- The critical section is enclosed with lock primitives

```
#pragma omp critical
{
    rdx = rdx + a;
    rdx2 = rdx2 * 2;
}
in__tone_set_lock(&in__tone_critical_lock_);
in__tone_global_flush();
{ rdx = rdx + a;
  rdx2 = rdx2 * 2;
}
in__tone_global_flush();
in__tone_unset_lock(&in__tone_critical_lock_);
```
The **atomic** construct

```c
#pragma omp atomic
rdx = rdx + foo();
```

- The atomic update is replaced with a call to the run-time which do the actual update atomically:

```c
... in__tone_atomic_update(&rdx,
        in_tone_type_f_float, in_tone_op_plus, foo());
...```

- Support for the final reduction of reduction variables is also implemented in a similar way

Summary about OdinMP/Balder

- OdinMP can be retrieved here: [http://www.odinmp.org](http://www.odinmp.org)
- Based on source-to-source transformations
- No magic transformations
- Subtle details in the OpenMP specification have required clarifications from the OpenMP Architecture Review Board (ARB)
  - Much better in OpenMP v2.5

Shared Memory on a Cluster

- It is possible to implement the illusion of a shared memory on a Cluster
  - Access control through code instrumentation
  - Page-based access control

Access Control Through Code Instrumentation

- Instrument load- and store-instructions to shared data with code that checks the current state (Modified, Exclusive, Shared, Invalid)
- Call a coherence handling routine if a load-access to a block in invalid state
- Call a coherence handling routine if a store-access to a block not in Modified state
- This approach can use any coherence granularity (byte, word, cache block... )
Page-based access control

*Software Distributed Shared Memory*

- User the virtual address memory protection mechanism for access control
- A valid address mapping means a valid page in local memory
- Add a *page-fault handler* that invokes protocol actions for invalid pages

---

**A Naive Software DSM Approach**

1 load

![Diagram showing Virtual addr space and Software DSM for load operation](image)

2 load

![Diagram showing Virtual addr space and Software DSM for load operation](image)

---

**A Naive Software DSM Approach**

3 store

![Diagram showing Virtual addr space and Software DSM for store operation](image)

---

**A Naive Software DSM Approach**

4 load

![Diagram showing Virtual addr space and Software DSM for load operation](image)

- Simple model, easy to implement, Sequentially consistent
- Performance problems: a word is needed but a whole page is fetched
Implementation in Unix

- Protection per page
  - PROT_READ
  - PROT_WRITE
  - PROT_EXEC
  - PROT_NONE
- System call to set protection: mprotect
- Example:
  - signal(SIGSEGV, page_fault_handler)
  - mprotect(pageA, 4K, PROT_NONE);
  - store to pageA
  - page fault (segmentation violation) → invocation of page fault handler
  - page fault handler requests page from the node that has it and sets protection using mprotect

Software DSM in TreadMarks

Solves the problem by:
- Lazy Release Consistency: Updates are not propagated to the user until after an acquire-operation (LOCK)
- Multiple-Writer protocol: Two (or more) processors may simultaneously modify a page if they modify disjunct regions of the page

LRC in TreadMarks

OpenMP on Clusters

- The Balder run-time system works on clusters of SMPs
  - Need additional calls to allocate shared memory
- The OdinMP translator can target the cluster-enabled run-time system
- Shared variables need to be allocated explicitly
  - Global variables are allocated separately
    - Variables are allocated and initialized at system startup
  - Shared stack variables are put on a stack in the shared address space
Summary

• Programming models and machine models are two different things
• The message-passing model is a relatively simple model to implement on both DM and SM machines
• MPI is a complex model containing 200+ functions
• However, only six functions are needed for most purposes
• Strong middleware support is needed for both MPI and OpenMP implementations
• It is possible to implement shared memory on top of clusters, but performance relies heavily on program behaviour