The Design of MPI Based Distributed Shared Memory Systems to Support OpenMP on Clusters

IEEE Cluster 2007, Austin, Texas, September 17-20

H’sien Jin Wong
Department of Computer Science
The Australian National University

Supervised by: Dr. Alistair P. Rendell
Overview

- Distributed Shared Memory and OpenMP
- Characteristics of our DSM
- Comparing Polling and Event-driven Models
- Flush Implementation
- Conclusions and Future Work
Distributed Shared Memory and OpenMP

- Distributed Shared Memory (DSM)
  - DSMs provide a means to use the shared memory programming paradigm across nodes of a cluster.
  - E.g. TreadMarks, SCASH

- OpenMP
  - OpenMP is a directive based approach for writing parallel programs.
  - The OpenMP style of parallelization is incremental.
  - OpenMP programming belongs to the shared memory programming paradigm.

- OpenMP implementation using DSMs
  - Using the appropriate compiler tools, OpenMP code can be compiled to make use of DSM libraries. This allows us to use OpenMP on clusters.
  - E.g. Intel CLOMP, ParADE, OdinMP, Omni/SCASH
Characteristics of our DSM

- **Page-based**: Shared memory is organized into pages. DSM pages can be any positive multiple of the system page size.

- **Home-based**: Each page is assigned a home process upon allocation. The home of a page is where the Master Copy of a page is maintained. All other instances of the page at other processes are copies.

- Uses MPI for communicating.
  - MPI provides portability. Available on virtually all clusters.
  - Support from vendors of high performance network hardware (e.g. Quadrics MPI).
  - There are support tools for MPI (e.g. visualization tools like Jumpshot).
Background Daemons

- Background daemons are used to implement one-sided operations (page fetch, locks, flushes, etc.).

**SCASH-MPI**

(Ojima et. al.)

- Computation Thread
- Communication Thread (polling)
- Work
- DSM
- Comm
- Pthread

**Our DSM**

- Worker Process
- DSM Process (event-driven)
- Work
- DSM
- Comm
- MPI Process

**POLLLING**

**EVENT-DRIVEN**

- MPI2’s RMA functions are not used because not all one-sided operations are RMA related.
Polling vs. Event-driven

- Compare using NPB3.2-OMP EP Class A on dual core AMD nodes. We use the NODExTHREAD configurations: 2x1, 4x1, 1x2, 2x2.
Where we’re up to now...

- Distributed Shared Memory and OpenMP
- Characteristics of our DSM
- Comparing Polling and Event-driven Models
- Flush Implementation
- Conclusions and Future Work
Implementing the OpenMP Flush

- **OpenMP Flush** – Ensures that the local view of memory is consistent with global memory. On a flush:
  - Local changes are written to global memory.
  - Remote changes that have been flushed is brought into local memory.

- A simple flush protocol that can be implemented in the DSM is:
  - **Diff** – Determine the differences made to the region and send them to the home node to be applied onto the Master Copy.
  - **Refresh** – Update the local copy with the latest state at the Master Copy.
Synchronization using OpenMP Flush

- The OpenMP Flush directive can be used for non-standard task synchronization.
- Used by NPB3.2-OMP LU to synchronize neighbouring OpenMP threads in its pipelined solution.
- A simple example from “Parallel Programming in OpenMP” – Chandra, 2001.

```c
INITIALIZE
flag = 0

PRODUCER THREAD CONSUMER THREAD
data = ... do
!$omp flush (data) !$omp flush (flag)
flag = 1 while (flag .ne. 1)
!$omp flush (flag) !$omp flush (data)
... = data
```
Producer Consumer Experiment

**INITIALIZE**
flag = 0

**PRODUCER THREAD**
data = ...
do
 !$omp flush (data)  !$omp flush (flag)
flag = 1

**CONSUMER THREAD**
while (flag .ne. 1)
 !$omp flush (flag)  !$omp flush (data)
... = data

PRODUCER

CONSUMER

DATA

MASTER COPY

FLAG

MASTER COPY
Producer Consumer Experiment

INITIALIZE
flag = 0

PRODUCER THREAD  CONSUMER THREAD
data = ... do
!$omp flush (data)  !$omp flush (flag)
flag = 1  while (flag .ne. 1)
!$omp flush (flag)  !$omp flush (data)
... = data

- Using **Diff-and-Refresh** described earlier, this example will experience a lot of network traffic.
Diff-and-Refresh Waiting
Observer Flush

- **Goal**
  - Eliminate redundant inter-node communication.

- **Design**
  - A region can have observers.
  - When a process receives a refresh of a region, that process becomes an observer of the region.
  - Observers will be notified of changes to the region.
  - After subscribing to a region, processes only need to request for a refresh if it had received a notification of changes to the region.
Consumer Becomes an Observer

flag has no changes

No change notice received for flag. Only within the node communication.

Busy-waiting

CONSUMER is observing flag
When Producer Flushes Flag

- Received change notice for flag. Send ack message.
- flag has changes. Request Refresh
- Send notices to observers and remove them from the list.
- All acks received. Send Flush Done to PRODUCER.
Performance Evaluation: Laplace

- A finite-difference approximation of the 2D Laplace Equation that uses a 5-point stencil operation.

- We only do 1 iteration.

- The memory access profile of this calculation is a series of writes to one grid and reads from another grid. In short, the useful work here is a memory touching exercise with some calculation.

- Problem size is a $8192 \times 8192$ grid.

- Total of 262112 pages touched.
Experiment Layout: With Busy-wait

- It is the slow down caused by the busy-waiting consumers on the producer that matters. Not the efficiency of the busy-wait loop!
Experiment Layout: Without Busy-wait

- It is the slow down caused by the busy-waiting consumers on the producer that matters. Not the efficiency of the busy-wait loop!
Results

<table>
<thead>
<tr>
<th></th>
<th>No Busy-wait</th>
<th>Diff-and-Refesh</th>
<th>Observer Flush</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>88</td>
<td>96</td>
<td>89</td>
</tr>
<tr>
<td>Slowdown</td>
<td>9%</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Interrupting Events</td>
<td>1506221</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- Observer flush reduces flush related communication in busy-wait loops to a deterministic maximum amount.

\[ 12 = (\text{RefreshRequest} + \text{AckResponse} + \text{RefreshRequest}) \times 4 \]
Conclusions and Future Work

• Conclusions
  • Using an event-driven daemon task is more efficient than a polling one.
  • The Observer Flush removes unnecessary inter-node communication when a region of memory has no changes but is flushed more than once (e.g. the busy-wait loop).
  • MPI-based DSMs can benefit from tools designed for MPI. e.g. the Jump-shot visualization tool.

• Future Work
  • Rethink job scope of foreground and daemon tasks – events are often blocking, meaning the worker process waits for the daemon process to do everything. *When there’s not much to do, the relative cost of “passing the buck” to the daemon becomes expensive.*
  • Hybrid paradigms – use MPI within OpenMP code to talk between *threads*. Can be used to implement the non-standard synchronization and avoid the use of busy-wait loops.
Acknowledgements

• I am grateful to Y. Ojima, M. Sato, and T. Boku for helpful discussions and providing source copies of SCASH-MPI and Omni; from which I have learnt a lot about DSM construction and OpenMP translation.

• Alexander Technology for providing the cluster hardware on which this development work took place.

• The Cluster 2007 organizers for giving me the opportunity to present my work.
Questions