Memory Coherence in Shared Virtual Memory System

ACM Transactions on Computer Science (TOCS), 1989
KAI LI Princeton University
PAUL HUDAK Yale University

Oct. 11th 2011

Kim Myung Sun, Lee Chung Hyeon
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
Basic concept

What is shared virtual memory?

- Provides a virtual address space that is shared among all processors
- Applications can use it just as they do a traditional virtual memory
- Not only “pages” data between physical memories and disks
  - Conventional virtual memory system
- But also “pages” data between the physical memories of the individual processors
  - Data can naturally migrate between processors on demand

What do the author really want to say in this paper?

- Main difficulty in building a shared virtual memory
- Memory coherence problem
Multiprocessor system

Introduction

Loosely coupled

- A single address space virtually shared
- Address space is partitioned into pages

Tightly coupled

- A single address space physically shared
- Multi cache schemes (cache consistent protocol)
- Embedded System

Shared virtual memory
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
Main features

- Memory mapping managers
  - Keeping the address space **coherent**
  - Views its local memory as a large cache

- Key goals
  - To allow processes of a program to execute on different processors in parallel
  - To improve the performance of parallel programs
    - # parallel processes
    - Shared data update ratio
    - Contention for the shared data
Characteristics

- Partitioned into *pages*
- RO pages can have many copies on each CPU’s memory
- WR pages can reside in only one CPU’s memory
- Memory reference can cause a *page fault*

Slightly different from page fault in Embedded system
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
Basic concept & design issues

- What is the meaning of memory coherence?
  - The value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

- Design issues
  - **Granularity** of the memory units (“page sizes”)
    - Sending large packets
      - is not much more expensive than small one
      - is the greater chance for contention than small one
    - 1K bytes is suitable? (right size is clearly application dependent)
  - Memory coherence **strategies**
    - Page synchronization and page ownership
Memory Coherence Problem

Memory coherence strategies

- Page synchronization
  - Invalidation
    - Invalidates all copies of the page when write page fault occurs
    - There is only one owner processor for each page
  - Write-broadcast
    - Writes to all copies of the page when write page fault occurs

- Page Ownership
  - Fixed
    - A page is always owned by the same processor
  - Dynamic
    - Centralized manager
    - Distributed manager
      - Fixed
      - Dynamic
Data structures for page table

- Access: indicates the accessibility to the page (nil/read/write)
- Copy set: processor number (read copies of the page)
- Lock: synchronizes multiple page faults by different processes on the same processor and synchronizes remote page requests

```plaintext
lock(PTable[p].lock):
  LOOP
    IF test-and-set the lock bit THEN
    EXIT;
    IF fail THEN queue this process;
unlock(PTable[p].lock):
  clear the lock bit;
  IF a process is waiting on the lock
  THEN
    resume the process;

invalidate( p, copy_set )
  for i in copy_set DO
    send an invalidation request to processor i;
```
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
A Monitor-Like Centralized Manager Algorithm

- Consisting of a **per page data structure** and provide **mutually exclusive access** to the data structure.
- Only one processor can be the “monitor”
- Only the monitor knows the **owner**

<table>
<thead>
<tr>
<th>Centralized Manager : Monitor</th>
<th>Each processors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Info</strong></td>
<td><strong>PTable</strong></td>
</tr>
<tr>
<td>owner</td>
<td>the most recent processor to have write access to it</td>
</tr>
<tr>
<td>copy set</td>
<td>lists all processors that have copies of the page</td>
</tr>
<tr>
<td>lock</td>
<td>used for synchronizing request to the page</td>
</tr>
</tbody>
</table>
Algorithm 1 \textit{MonitorCentralManager}

\textbf{Read fault handler:}
\begin{verbatim}
Lock( PTable[ p ].lock );
IF I am manager THEN BEGIN
  Lock( Info[ p ].lock );
  Info[ p ].copyset := Info[ p ].copyset \cup \{ManagerNode\};
  receive page p from Info[ p ].owner;
  Unlock( Info[ p ].lock );
END;
ELSE BEGIN
  ask manager for read access to p and a copy of p;
  receive p;
  send confirmation to manager;
END;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
\end{verbatim}

\textbf{Read server:}
\begin{verbatim}
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
  PTable[ p ].access := read;
  send copy of p;
END;
Unlock( PTable[ p ].lock );
IF I am manager THEN BEGIN
  Lock( Info[ p ].lock );
  Info[ p ].copyset := Info[ p ].copyset \cup \{RequestNode\};
  ask Info[ p ].owner to send copy of p to RequestNode;
  receive confirmation from RequestNode;
  Unlock( Info[ p ].lock );
END;
\end{verbatim}

\textbf{Write fault handler:}
\begin{verbatim}
Lock( PTable[ p ].lock );
IF I am manager THEN BEGIN
  Lock( Info[ p ].lock );
  Invalidate( p, Info[ p ].copyset );
  Info[ p ].copyset := \{\};
  Unlock( Info[ p ].lock );
END;
ELSE BEGIN
  ask manager for write access to p;
  receive p;
  send confirmation to manager;
END;
PTable[ p ].access := write;
Unlock( PTable[ p ].lock );
\end{verbatim}

\textbf{Write server:}
\begin{verbatim}
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
  send copy of p;
  PTable[ p ].access := nil;
END;
Unlock( PTable[ p ].lock );
IF I am manager THEN BEGIN
  Lock( Info[ p ].lock );
  Invalidate( p, Info[ p ].copyset );
  Info[ p ].copyset := \{\};
  ask Info[ p ].owner to send p to RequestNode;
  receive confirmation from RequestNode;
  Unlock( Info[ p ].lock );
END;
\end{verbatim}
Centralized manager algorithms

- e.g. [Read page fault in P2]
Centralized manager algorithms

- e.g. [Read page fault in P2]

![Diagram showing the process of a request from P2 to P1 and the centralized manager algorithms.]

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
</tr>
<tr>
<td>lock</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Request
Centralized manager algorithms

- e.g. [Read page fault in P2]

**P1, Centralized Manager**

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P2 }</td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
</tr>
</tbody>
</table>

1. Request

P2

<table>
<thead>
<tr>
<th>PTable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
</tr>
</tbody>
</table>

P3

<table>
<thead>
<tr>
<th>PTable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>R</td>
</tr>
<tr>
<td>lock</td>
<td>0</td>
</tr>
</tbody>
</table>
Centralized manager algorithms

- e.g. [Read page fault in P2]

1. Request
2. Ask to send copy to P2

**P1, Centralized Manager**

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

**P2**

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

**P3**

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>
Centralized manager algorithms

- e.g. [Read page fault in P2]

P1, Centralized Manager

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

1. Request

2. Ask to send copy to P2

P2

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

P3

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

Seoul National University
Centralized manager algorithms

- e.g. [Read page fault in P2]

### P1, Centralized Manager

1. Request
2. Ask to send copy to P2
3. Send copy

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P2 }</td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable access</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>R</td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
</tr>
</tbody>
</table>
- e.g. [Read page fault in P2]

**Centralized manager algorithms**

1. Request
2. Ask to send copy to P2
3. Send copy
4. Confirmation
Centralized manager algorithms

- e.g. [Read page fault in P2]

1. Request
2. Ask to send copy to P2
3. Send copy
4. Confirmation

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P2 }</td>
</tr>
<tr>
<td>lock</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

Seoul National University
Centralized manager algorithms

- e.g. [Write page fault in P2]

### P1, Centralized Manager

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>
Centralized manager algorithms

- e.g. [Write page fault in P2]

1. Write Request

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P2 }</td>
</tr>
<tr>
<td>lock</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>R</td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
</tr>
</tbody>
</table>
Centralized manager algorithms

- e.g. [Write page fault in P2]

1. Write Request
2. Invalidate of copy set
Centralized manager algorithms

- e.g. [Write page fault in P2]

1. Write Request
2. Invalidate of copy set
3. Ask to send page to P2

P1, Centralized Manager

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

PTable

<table>
<thead>
<tr>
<th>access</th>
<th>lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

P2

PTable

<table>
<thead>
<tr>
<th>access</th>
<th>lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

P3

PTable

<table>
<thead>
<tr>
<th>access</th>
<th>lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Seoul National University
Centralized manager algorithms

- e.g. [Write page fault in P2]

1. Write Request
2. Invalidate of copy set
3. Ask to send page to P2
4. Send page to P2
Centralized manager algorithms

- e.g. [Write page fault in P2]

1. Write Request
2. P1, Centralized Manager
3. Ask to send page to P2
4. Send page to P2
5. confirmation

<table>
<thead>
<tr>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>lock</td>
</tr>
</tbody>
</table>
The primary difference from the previous one

- Synchronization of page ownership move to the individual owners
- The locking mechanism on each processor now deals not only with multiple local requests, but also with remote requests
- The copy set field is valid iff the processor that holds the page table is the owner of the page.
- But, for large $N$ there still might be a bottleneck at the manager processor because it must respond to every page fault
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
Distributed manager algorithms

A Fixed Distributed Manager Algorithm

- Predetermined subset of the pages
  - e.g.,
    - \( H(p) = p \mod N \), \( N \) is # of processors
    - \( H(p) = (p/s) \mod N \), \( s \) is # of pages per segment (e.g., \( s = 3 \))
  - Proceeds as in the centralized manager algorithm
  - Superior to the centralized manager algorithms when a parallel program exhibits a high rate of page faults
  - To find the good \( H(p) \) that fits all Appl. well is difficult
A broadcast distributed manager algorithm

- Each processor manages precisely those pages that it owns.
- Faulting processor send broadcasts into the network to find the true owner of a page.
- All broadcast operations are atomic.
- All processors have to process each broadcast request → slowing down the system.
Distributed manager algorithms

e.g. [Read page fault in P2]

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>owner</td>
<td></td>
<td>P3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2</th>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td></td>
</tr>
<tr>
<td>owner</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3</th>
<th>PTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
</tr>
<tr>
<td>owner</td>
<td>P3</td>
</tr>
</tbody>
</table>
Distributed manager algorithms

e.g. [Read page fault in P2]

1. Broadcast read request
Distributed manager algorithms

e.g. [Read page fault in P2]
A Dynamic Distributed Manager Algorithm

- Keeping track of the ownership of all pages in each processors’ local Ptable
  - There is no fixed owner or manager
  - Owner field -> probOwner field
  - The information that it contains is just a hint
  - When a processor has a page fault, it sends request to the processor indicated by the probOwner field
    - If that processor is the true owner, it proceeds as in the centralized manager algorithm
    - Else, forwards the request to the processor indicated by its probOwner field
Distributed manager algorithms

Dynamic Distributed Manager

Read fault handler:

\[
\text{Lock( PTable[ p ].lock );}
\]
\[
\text{ask PTable[ p ].probOwner for read access to p;}
\]
\[
PTable[ p ].probOwner := \text{self;}
\]
\[
PTable[ p ].access := \text{read;}
\]
\[
\text{Unlock( PTable[ p ].lock );}
\]

Read server:

\[
\text{Lock( PTable[ p ].lock );}
\]
\[
\text{IF I am owner THEN BEGIN}
\]
\[
PTable[ p ].copyset := PTable[ p ].copyset \cup \{ \text{Self}\};
\]
\[
PTable[ p ].access := \text{read;}
\]
\[
send p and PTable[ p ].copyset;
\]
\[
PTable[ p ].probOwner := \text{RequestNode;}
\]
\[
\text{END}
\]
\[
\text{ELSE BEGIN}
\]
\[
\text{forward request to PTable[ p ].probOwner;}
\]
\[
PTable[ p ].probOwner := \text{RequestNode;}
\]
\[
\text{END;}
\]
\[
\text{Unlock( PTable[ p ].lock );}
\]

Write fault handler:

\[
\text{Lock( PTable[ p ].lock );}
\]
\[
\text{ask PTable[ p ].probOwner for write access to page p;}
\]
\[
\text{Invalidate( p, PTable[ p ].copyset );}
\]
\[
PTable[ p ].probOwner := \text{self;}
\]
\[
PTable[ p ].access := \text{write;}
\]
\[
PTable[ p ].copyset := \{};
\]
\[
\text{Unlock( PTable[ p ].lock );}
\]

Write server:

\[
\text{Lock( PTable[ p ].lock );}
\]
\[
\text{IF I am owner THEN BEGIN}
\]
\[
PTable[ p ].access := \text{nil;}
\]
\[
send p and PTable[ p ].copyset;
\]
\[
PTable[ p ].probOwner := \text{RequestNode;}
\]
\[
\text{END}
\]
\[
\text{ELSE BEGIN}
\]
\[
\text{forward request to PTable[ p ].probOwner;}
\]
\[
PTable[ p ].probOwner := \text{RequestNode;}
\]
\[
\text{END;}
\]
\[
\text{Unlock( PTable[ p ].lock );}
\]

w/o manager
Distributed manager algorithms

- e.g., [Read page fault in P2]

<table>
<thead>
<tr>
<th>PTable access</th>
<th>P1 access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTable lock</td>
<td>P1 lock</td>
<td></td>
</tr>
<tr>
<td>PTable copy set</td>
<td>P1 copy set</td>
<td>{ P1 }</td>
</tr>
<tr>
<td>probOwner P3</td>
<td>probOwner P3</td>
<td></td>
</tr>
</tbody>
</table>

- P2:
  | PTable access | R |
  | PTable lock   |   |
  | PTable copy set | { P1 } |
  | probOwner P1 |   |

- P3:
  | PTable access | R |
  | PTable lock   |   |
  | PTable copy set | { P1 } |
  | probOwner P3 |   |
Distributed manager algorithms

- e.g., [Read page fault in P2]

<table>
<thead>
<tr>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
</tr>
<tr>
<td>probOwner</td>
<td>P3</td>
</tr>
</tbody>
</table>

1. read request
Distributed manager algorithms

- e.g., [Read page fault in P2]

1. read request
2. forward request

Need not send a reply to the requesting processor
Distributed manager algorithms

- e.g., [Read page fault in P2]

1. read request
2. forward request
3. Send copy of p and copy set
Distributed manager algorithms

- e.g. [Write page fault in P1]

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
<td></td>
</tr>
<tr>
<td>probOwner</td>
<td>P2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTable</td>
</tr>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>probOwner</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTable</td>
</tr>
<tr>
<td>access</td>
</tr>
<tr>
<td>lock</td>
</tr>
<tr>
<td>copy set</td>
</tr>
<tr>
<td>probOwner</td>
</tr>
</tbody>
</table>
Distributed manager algorithms

- e.g. [Write page fault in P1]

1. write request

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
<td></td>
</tr>
<tr>
<td>probOwner</td>
<td>P2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1 }</td>
<td></td>
</tr>
<tr>
<td>probOwner</td>
<td>P2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P3 }</td>
<td></td>
</tr>
<tr>
<td>probOwner</td>
<td>P2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th>access</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy set</td>
<td>{ P1, P3 }</td>
<td></td>
</tr>
<tr>
<td>probOwner</td>
<td>P2</td>
<td></td>
</tr>
</tbody>
</table>
Distributed manager algorithms

• e.g. [Write page fault in P1]

1. write request
2. Send p and copy set
Distributed manager algorithms

- e.g. [Write page fault in P1]
Distributed manager algorithms

- e.g. [Write page fault in P1]

1. write request
2. Send p and copy set
3. Invalidate in copy set
Distribution of Copy Set

- Copy set data is distributed and stored as a tree

- Improves system performance
  - “Divide and conquer” effect
    - Balanced tree $\rightarrow \log m$ (faster invalidation) for $m$ read copies

- Owner
- ProbOwner

Write fault invalidation through Copy set

Read fault collapse through ProbOwner
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
Experiments

Experimental Setup

- Implement a prototype shared virtual memory system called IVY (Integrated shared Virtual memory at Yale)
  - Implemented on top of a modified Aegis operating system of Apollo DOMAIN computer system
  - IVY can be used to run parallel programs on any number of processors on an Apollo ring network

Parallel Programs

- Parallel Jacobi program
  - Solving three dimensional PDEs (partial differential equations)
- Parallel sorting
- Parallel matrix multiplication
- Parallel dot-product
The speedup of a program is the ratio of the execution time of the program on a single processor to that on the shared virtual memory system. Experimental Setup

All the speedups are compared against the ideal
Experiments

- Speed up of 3D-PDE, where \( n = 50^3 \)

- Super-linear speed up
  - Data structure for the problem is greater than the size of physical memory on a single processor

Fig. 5. Speedups of a 3-D PDE where \( n = 50^3 \).
Experiments

- Disk paging

Data structure for the problem > Phy. Mem.
- As if combined physical memory by SVM.

Fig. 6. Disk paging on one processor and two processors.
Experiments

- Speed up of 3-D PDE, where $n=40^3$

*Fig. 7. Speedups of a 3-D PDE where $n = 40^3$."

- Data structure for the problem < Phy. Mem
Experiments

- Speedup of the merge-split sort, where 200K elements
  - Ideal solution
    - Even with no communication costs, merge-split sort does not yield linear speed up
    - the costs of all memory references are the same

![Graph showing speedup of merging sort with n processors](image)

**Ideal Solution**

**Experimental Result**

Fig. 8. Speedup of the merge-split sort.
Speed up of parallel dot-product where each vector has 128K elements.

Shared virtual memory system is not likely to provide speed up
  • Ratio of the communication cost to the computation cost is large.
Experiments

- Speed up of matrix multiplication, where 128 x 128 square matrix

\[ C = AB \]

Program exhibits a high degree of localized computation

Fig. 10. Speedup of the matrix multiplication program.
Compare to memory coherence algorithms

- All algorithms have similar number of page faults
- Number or requests of fixed distributed manager algorithm is similar to improved centralized manager algorithm
  - Both algorithm need a request to locate the owner of the page for almost every page fault that occurs on a non-manager processor

Experiments

Overhead = Messages = requests

ProbOwner fields usually give correct hints
Agenda

- Introduction
- Shared virtual memory
- Memory coherence problem
- Centralized manager algorithms
- Distributed manager algorithms
- Experiments
- Conclusion
This paper studied two general classes of algorithms for solving the memory coherence problem:

- Centralized manager
  - Straightforward and easy to implement
  - Traffic bottleneck at the central manager

- Distributed Manager
  - Fixed dist. Manager
    - Alleviates the bottleneck
  - Dynamic dist. Manager
    - The most desirable overall features

Scalability and granularity
- Over 8 processors? And other than 1 KB?
Questions or Comments?