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

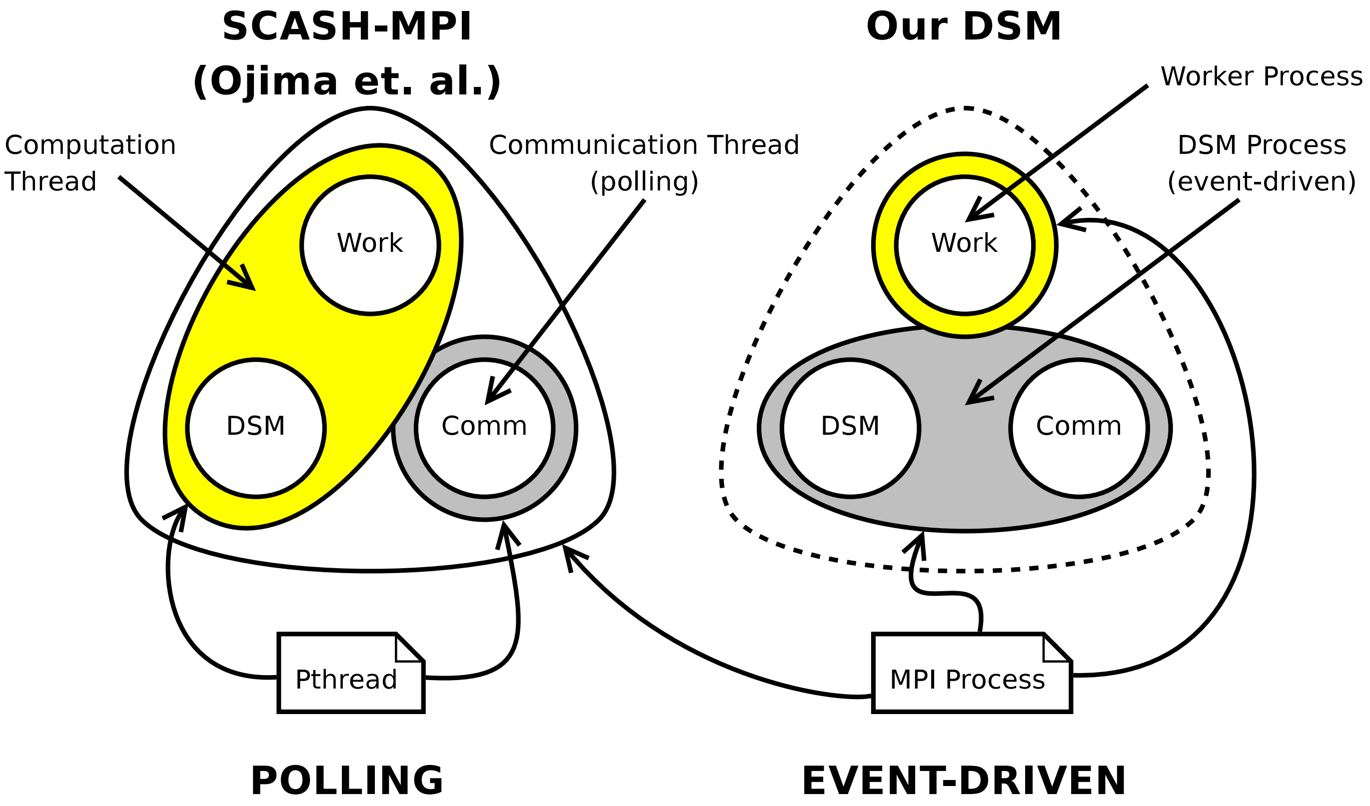
- Distributed Shared Memory (DSM)
  - DSMs provide a means to use the shared memory programming paradigm across nodes of a cluster.
  - E.g. TreadMarks, SCASH
- OpenMP
  - OpenMP is a directive based approach for writing parallel programs.
  - The OpenMP style of parallalization is incremental.
  - OpenMP 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 libaries. 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

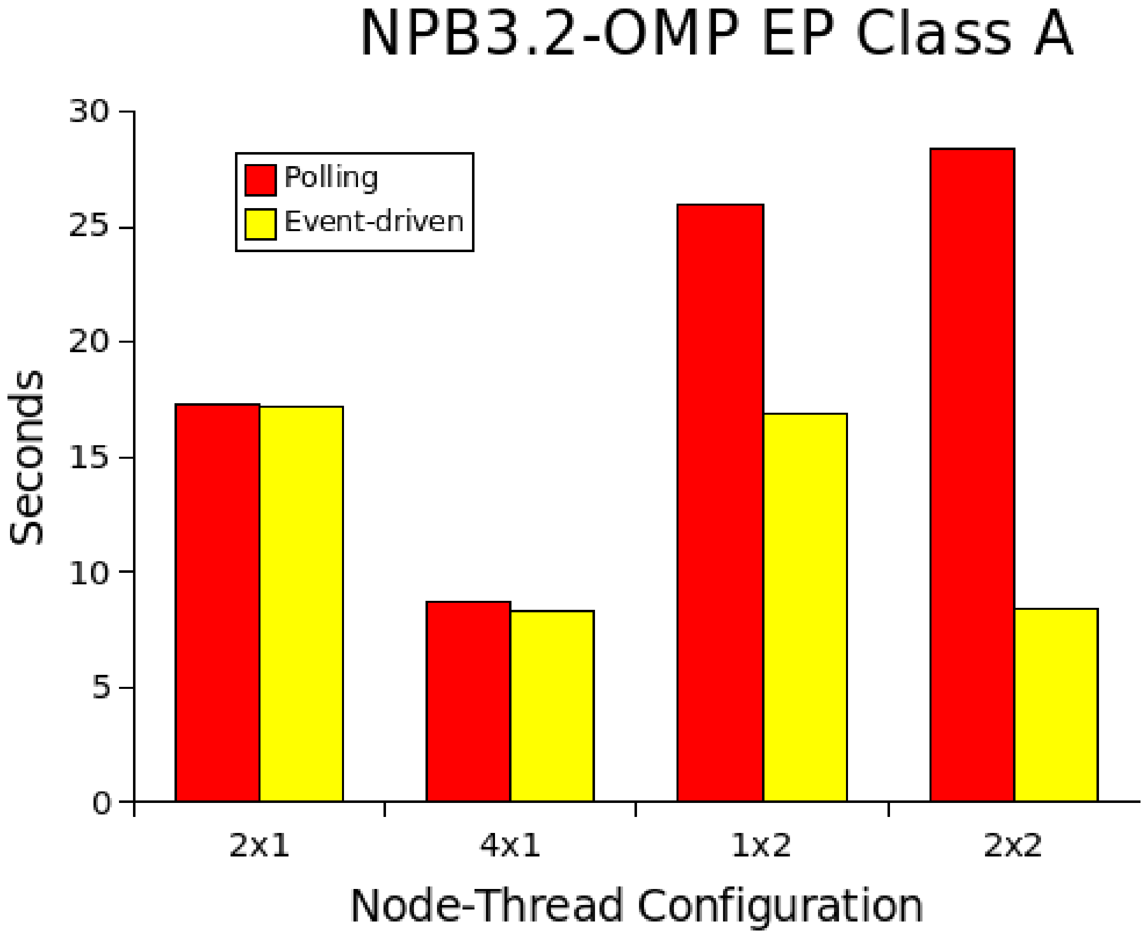
- Background daemons are used to implement one-sided operations (page fetch, locks, flushes, etc.).



- 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 Progamming in OpenMP” – Chandra, 2001.

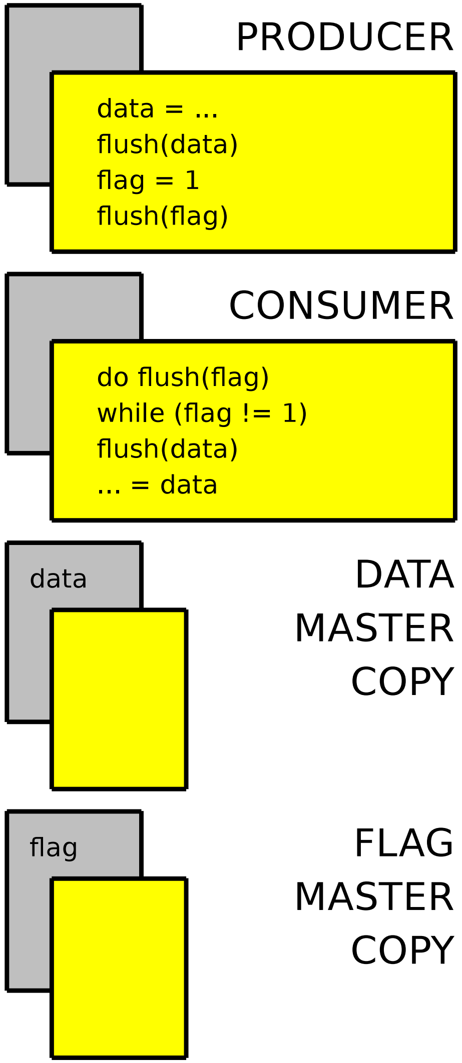
```
INITIALIZE
flag = 0

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



PRODUCER

data = ...
flush(data)
flag = 1
flush(flag)

CONSUMER

do flush(flag)
while (flag != 1)
flush(data)
... = data

data

DATA
MASTER
COPY

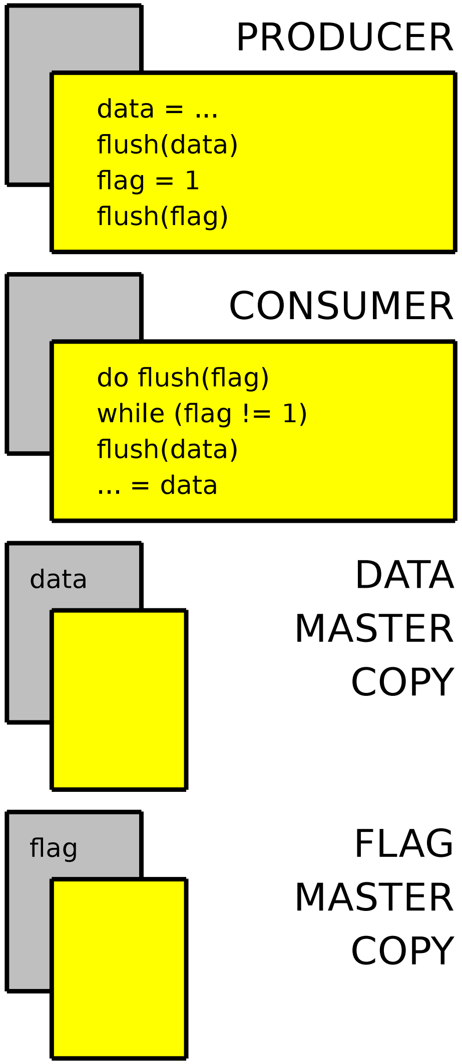
flag

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
```



PRODUCER

```
data = ...
flush(data)
flag = 1
flush(flag)
```

CONSUMER

```
do flush(flag)
while (flag != 1)
flush(data)
... = data
```

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

Diff-and-Refresh Waiting

PRODUCER

CONSUMER

DATA

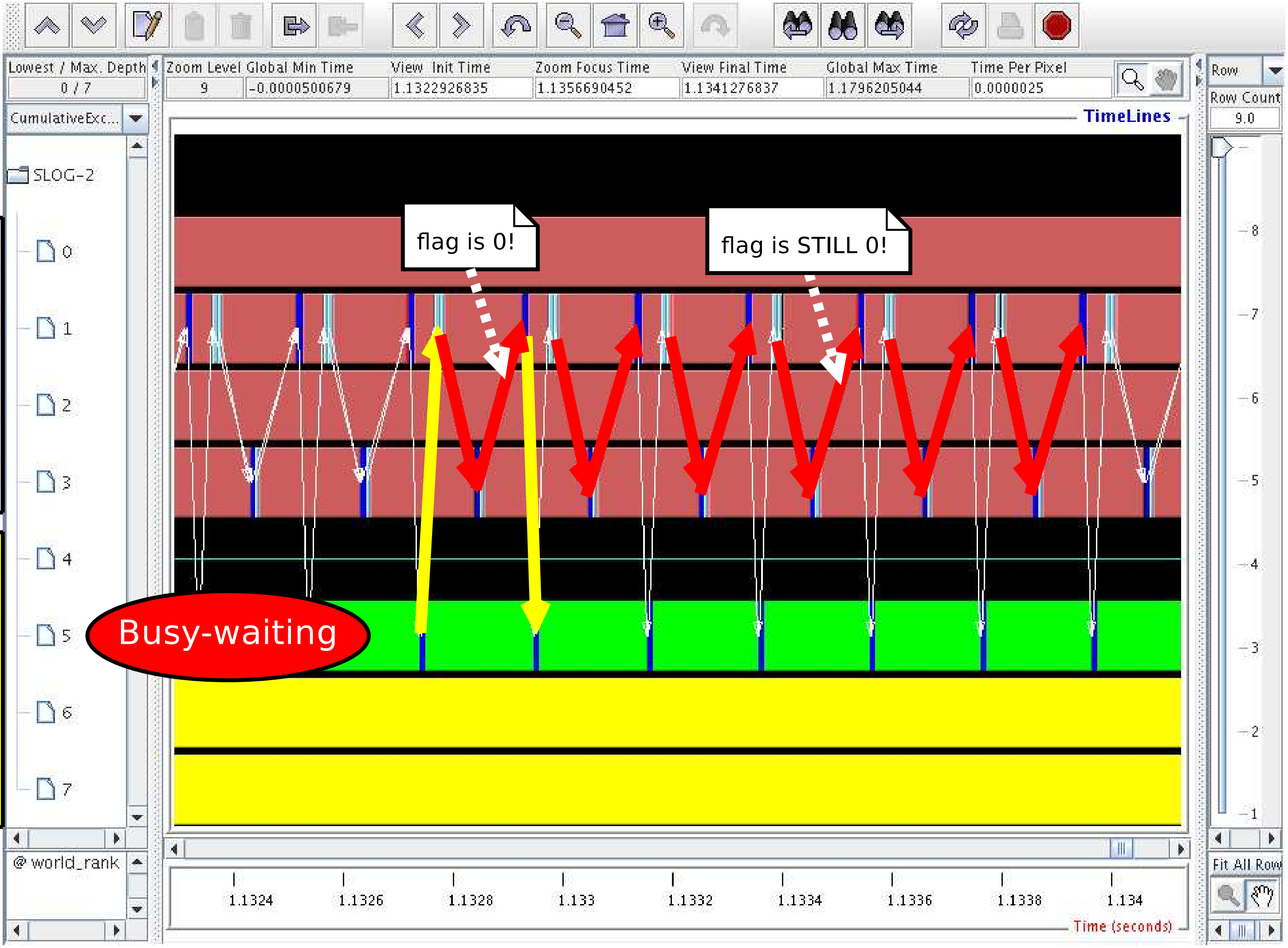
FLAG

PRODUCER

CONSUMER

DATA

FLAG



Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 7	9	-0.0000500679	1.1322926835	1.1356690452	1.1341276837	1.1796205044	0.0000025

Row Count: 9.0

flag is 0!

flag is STILL 0!

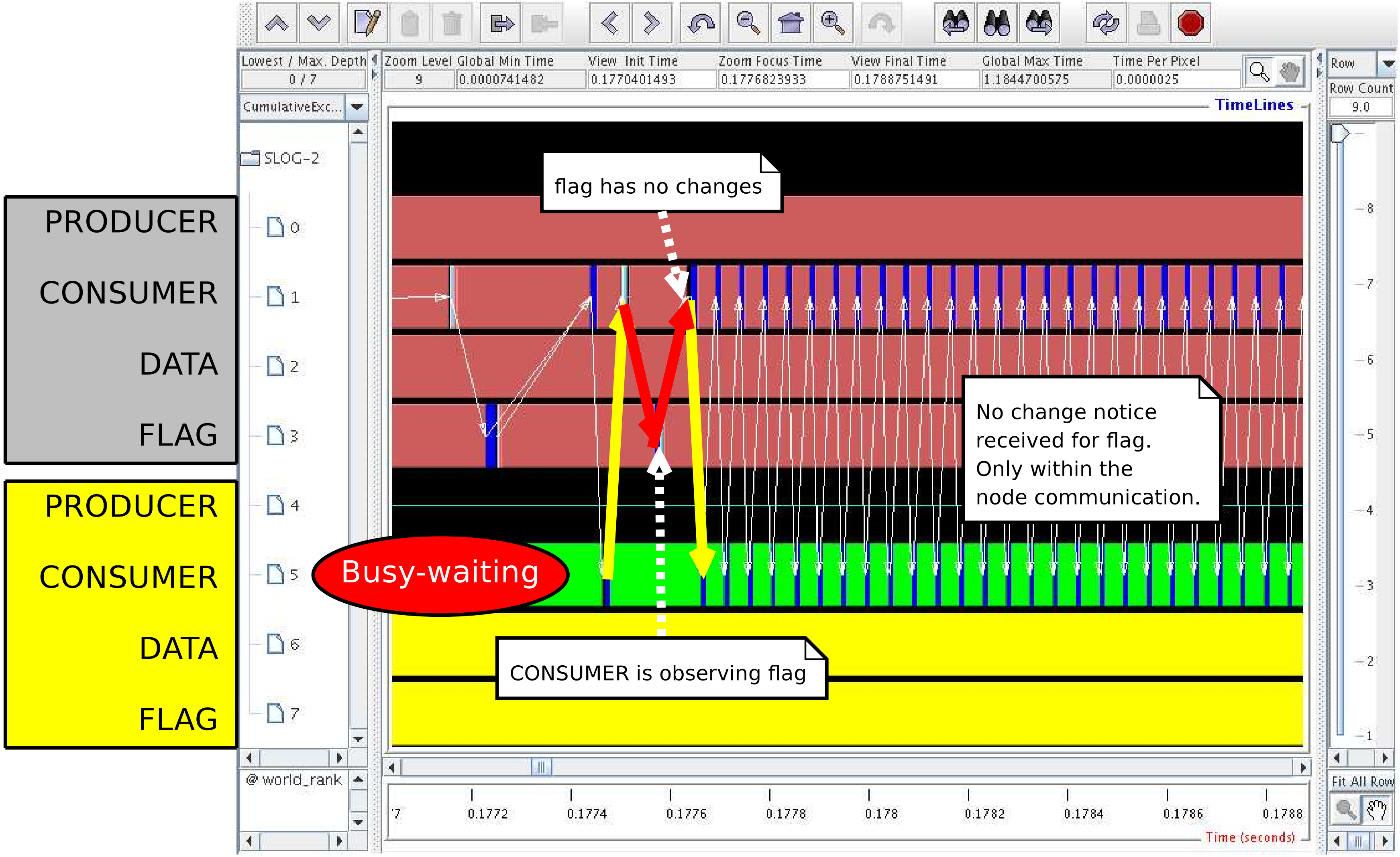
Busy-waiting

THE AUSTRALIAN NATIONAL UNIVERSITY

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



PRODUCER
CONSUMER
DATA
FLAG

PRODUCER
CONSUMER
DATA
FLAG

Lowest / Max. Depth: 0 / 7
Zoom Level: 9
Global Min Time: 0.0000741482
View Init Time: 0.1770401493
Zoom Focus Time: 0.1776823933
View Final Time: 0.1788751491
Global Max Time: 1.1844700575
Time Per Pixel: 0.0000025
Row Count: 9.0

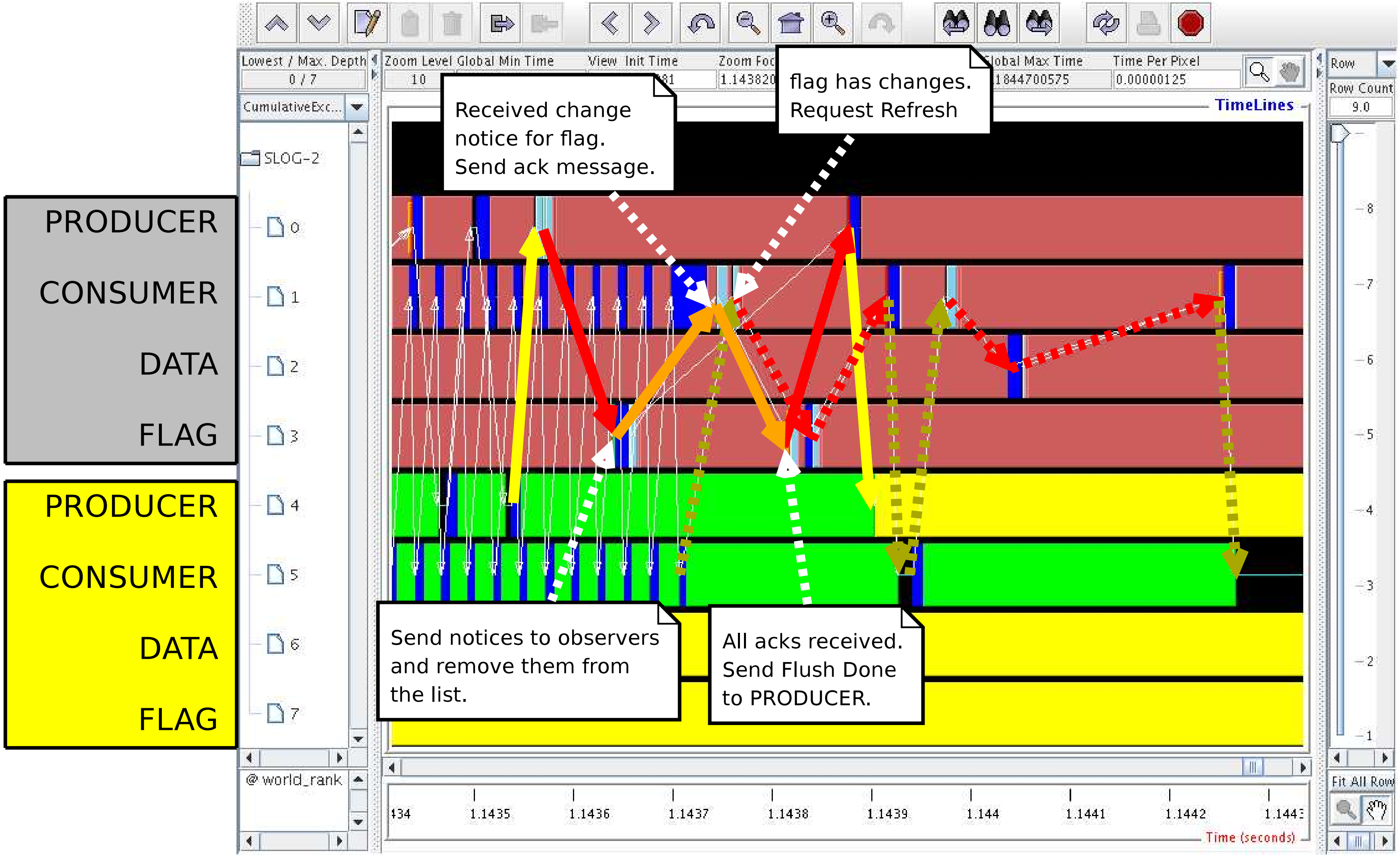
flag has no changes

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.

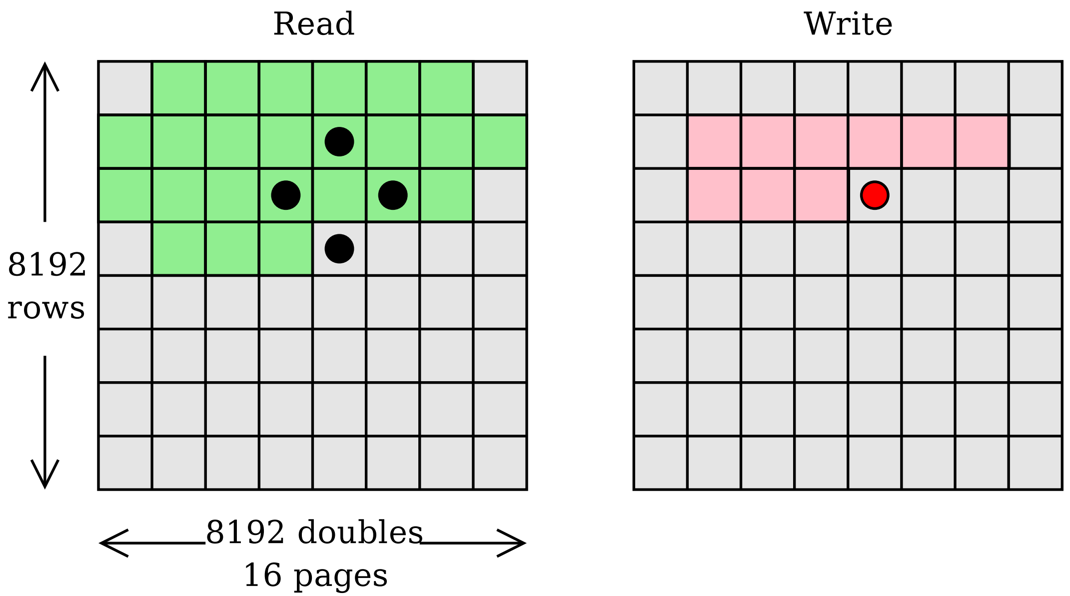
flag has changes. Request Refresh

Send notices to observers and remove them from the list.

All acks received. Send Flush Done to PRODUCER.

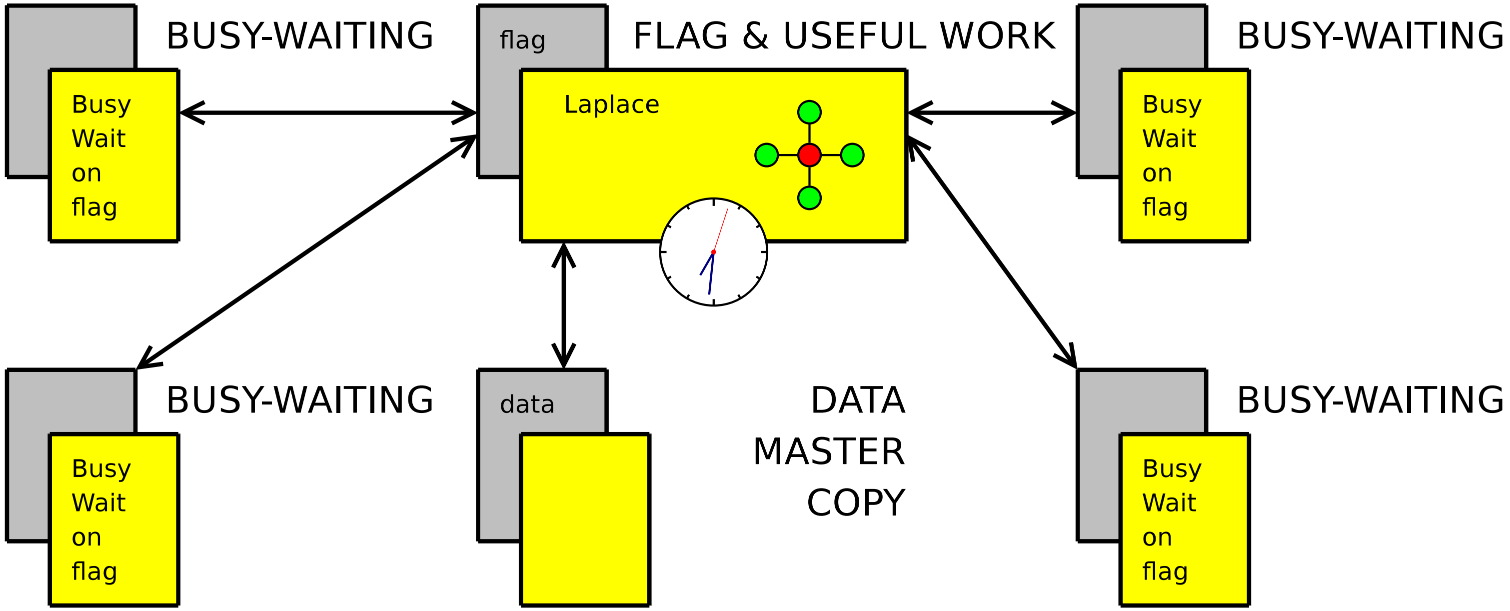
Performance Evaluation: Laplace

- 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 8192×8192 grid.
- Total of 262112 pages touched.



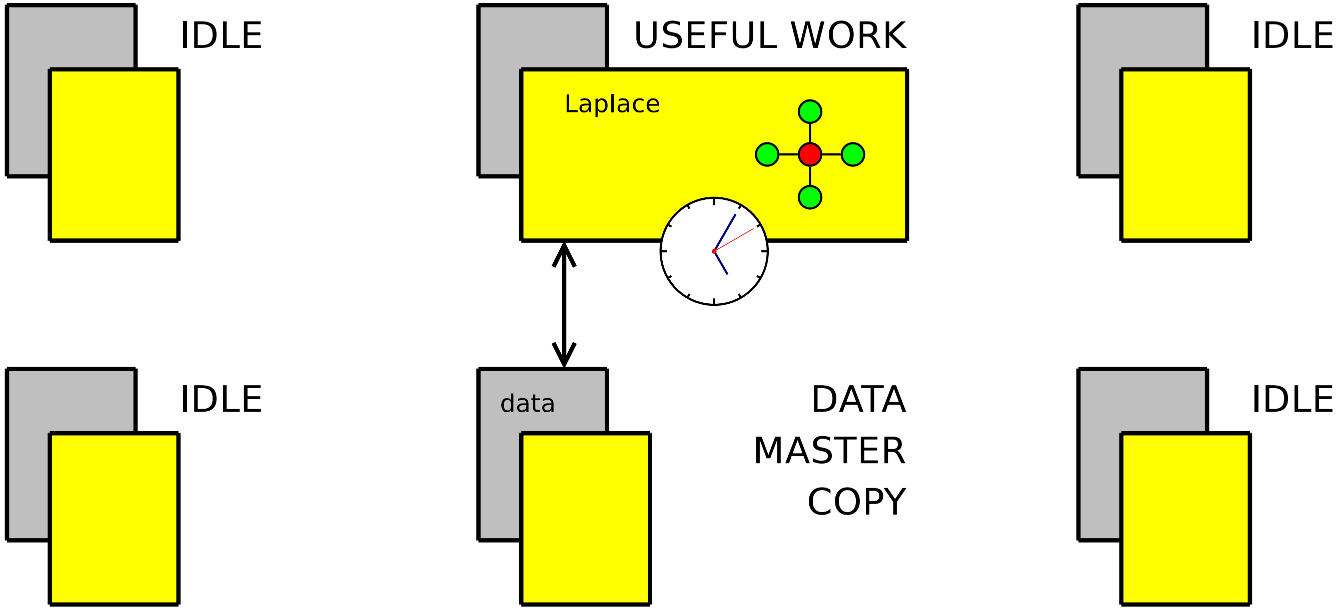
Experiment Layout: With Busy-wait

- 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

	No Busy-wait	**Diff-and-Refesh**	**Observer Flush**
Seconds	88	96	89
Slowdown		9%	1%
Interrupting Events		1506221	12

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

$$12 = (\mathrm{RefreshRequest} + \mathrm{AckResponse} + \mathrm{RefreshRequest}) \times 4$$

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 Jumpshot 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