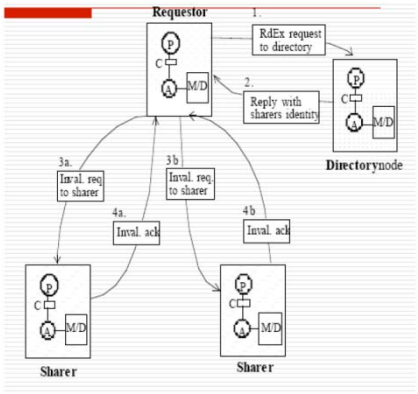
Question 1: Proposal

Name of Participants: Simon Foucher (260 223 197)
WebCT Group #: ECSE-420 07

Abstract:

Sudoku puzzles are a challenging number game that is gaining much popularity. Solving such puzzles by hand can be a tedious task. This project proposes to design and implement a parallel computing application that solves Sudoku puzzles. The code will use a multi-threading environment to solve the puzzle. The work will be divided in 27 work segments: analyzing the rows, the columns and the sub-squares. Upon starting, all 27 work segments will be added to a work queue. Upon becoming available, every thread will dequeue a work load and attempt to solve some component it contains. If it modifies any components, the work segment (row, column or sub square) will be re-added in the queue for further processing (in case this new information can help solve other elements in the segment). Once the queue is empty, the program will print out the result or an error message if some elements are still unsolved.

QUESTION 2.



a) Write-through invalidate directory-based machine for a write operation by a processor i, in the case when the dirty bit is ON.

1. Requestor (Pi) sends RdEx request to directory
2. Directory replies with ID of all the processors sharing the cache line, update itself by marking all those entries as invalid and Pi with valid for that cache line
3. Requestor (Pi) sends Invalid request for the cache line it wants to write to all the sharers returned by the directory and wait for their ack.
4. All the sharing processors eventually send an Invalidate Acknowledge message back to the Requestor (Pi)
5. The Requestor (Pi) can now write his cache with the new data.
6. Because we are implementing write through, Pi also updates the memory.

b) Describe what happens with all the required transactions when considering the update protocol.

Assuming we now have a write-through update directory based machine, here is the new flow of events:

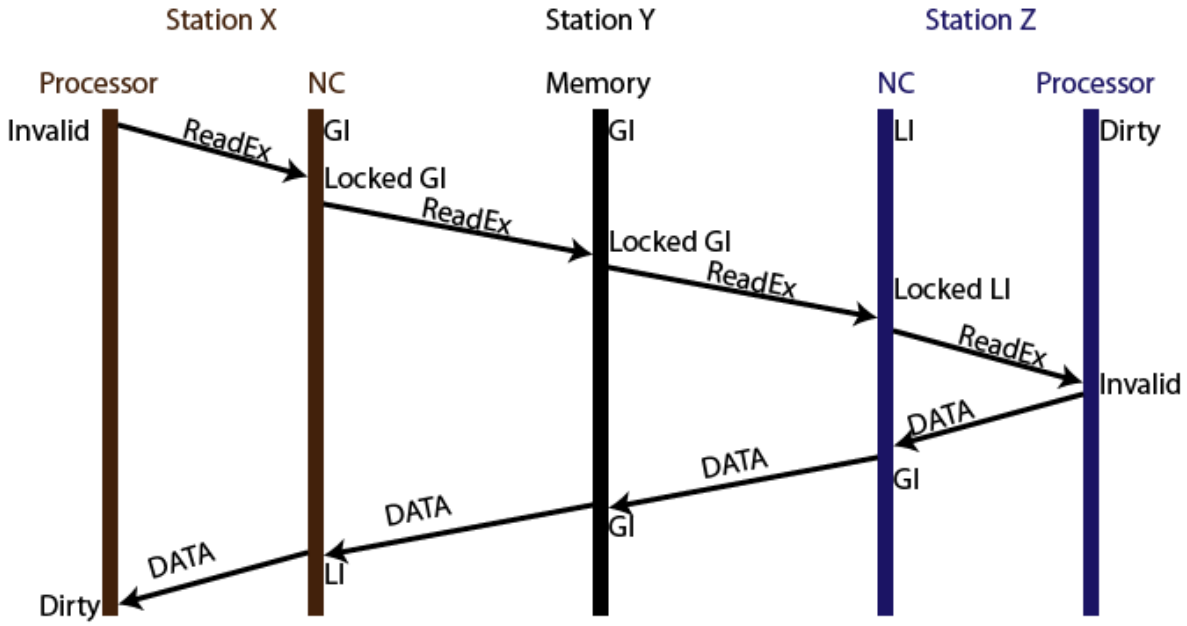
1. Requestor (Pi) sends RdEx request to directory
2. Directory replies with ID of all the processors sharing the cache line and get updated to include Pi as possessing a copy of the data
3. The Requestor (Pi) writes to his cache with the new data.
4. Requestor (Pi) sends new dada for the cache line it just modified to all the sharers returned by the directory. All the sharing processors now hold the new data
5. Because we are implementing write through, Pi also updates the memory.

3. For NUMachine distributed directory-based cache coherence protocol, show the timing diagram for local exclusive read request to the cache line that is in GI state in the main memory.

GI state in the memory module indicates that there exists a remote network cache (identified by the routing mask) with a copy in LV or LI state and that there is no valid copy in the NC or in any of the processor on the station. Weather the remote cache line was in LV or LI state, if it encounters a Remote ReadEx, it will fall in the state GI. In the LV state, the memory (or network cache) as well as the secondary caches indicated by the processor mask have a valid copy. In the LI state, only one of the local secondary caches has a copy

For this example:

- X- Processor launching the RdEx command.
- Y- Home memory location of the requested Cache line in GI state
- Z- Processor currently holding the latest version of the data



Flow:

1. X processor's request goes first to the network cache (NC) on station X, currently in GI state
2. The network cache locks itself in GI state and forwards the ReadEx request packet to Station Y (home location of the data), currently in GI state
3. Station Y is also locked in GI state and the ReadRx is propagated to Station Z; currently holding the latest version of the data in LI state.
4. Station Z's NC is locked in LI state and ReadEx is sent to processor Z, currently holding the data in dirty state.
5. Processor Z sets its cache line to dirty then forward the data to its NC.
6. Station Z's NC gets the data, forwards it to the upper ring and updates its state to GI
7. Station Y ignores the data on the upper ring, but when it sees it, unlocks itself and stays in GI state. (it needs not to record the updated data value since it was a ReadEx from elsewhere)
8. Station X's NC picks up the data and forwards it to Processor X, then releases its state to LI
9. Processer X pikes up the data and sets its cache line to dirty.

(ref used: http://www.eecg.toronto.edu/parallel/parallel/numachin.hier/node5.html)