Consistency of Shared Memory and the Mermera Programming Model

George Mark Ruban

gmn@cs.bu.edu

29 March 1995

Abstract

A shared memory model for distributed computing is easier to program for than one that relies on message passing, but ensuring shared memory consistency takes much overhead that is unnecessary for many algorithms. This has prompted the study and implementation of less consistent models for shared memory, many of which were formalized and classified in a paper by Professor Abdelsalam Heddaya and Himanshu Sinha, both of Boston University. The same authors are also responsible for *Mermera*, a programming model that allows the use of different consistency levels of shared memory for a distributed computing program. A second paper concentrated on Mermera.

In this paper I plan to informally summarize Heddaya and Sinha's papers, formally classifying and relating different consistency levels of shared memory, and describing how Mermera implements each. I will also add four different relaxed consistency levels of shared memory described by Kai Hwang in his book, *Advanced Computer Architecture*, describe each, informally relate each to the models and classification of Heddaya and Sinha, and explain how the Mermera program could implement each.

Shared Memory in Parallel Computing

Even though computer processor speeds are growing with every year, the size and the difficulty of the problems computers are being asked to solve is growing even faster. As this trend seems sure to continue, more work is being done on parallel computing, letting several processors, or even several computers, work together on the same problem. In theory, many smaller, cheaper computers working together might be able to do the job of a giant mainframe. In practice, of course, writing programs for many processors working together is harder than just writing for one, and writing for entirely different computers is even harder.

Any parallel program needs a way for its different processors to communicate, and most popularly this has been done through the message passing model. A processor that wants to send a message picks a target and sends an electronic packet of information. The receiving processor must be ready to receive; it must read that information before more is sent, or a message might be lost. This requires extra care on the part of the programmer, to make sure that processors regularly check for messages, to anticipate which process will be executing which piece of code on which processor when, and so what other processes it has to communicate with, and where they are. This requires so much care, in fact, that it has been referred to as a "hardship" for the programmer.[1]

The second most popular model for parallel computer inter-communication is shared memory, where the programmer can assume that all the processors are really working on the same data, so when a change is made by one processor it can be read by all. This means that processes can have any information just by reading it. The advantages of a shared memory model are in easier programming: the programmer does not have to worry about the way in which the processes are physically connected, and only rarely about how many processors or processes are actually executing the code. This

---

[1] Kai Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993. p. 26.

allows programs to be more portable between different architectures in a way that message-passing programs are not. For example, a message passing program written on a rectangular mesh architecture might rely on passing an item of data from each processor to the one directly "north" of it -- and would not be implementable on a shared bus network of workstations, where there was no concept of "north." The equivalent shared memory program would simply have each process read the data from, or write it to memory as needed, and would not rely on just how the data was delivered, or where each process was running.

The problem with the ideal of shared memory is that the implementation is usually not on a machine with one central memory equally accessible by all processors (called UMA -- a Uniform Memory Access machine) but rather memory that is cached by individual processors, partially (NUMA - Non-Uniform Memory Access), or completely (COMA - Cache Only Memory Access), so that memory is "closer" to, and faster to access by some processors than others.[2] Growing even more popular is a *distributed* or *multi-computer* implementation, a system of essentially separate machines linked by some message passing network. Each memory location "belongs" to a particular processor, which is the only one that can access it directly. The reason for the popularity of this is that such a system is more easily scalable -- the more self-sufficient a node, with processor, memory, input, and output, is, the easier it is to add nodes to or subtract nodes from the system -- and each workstation or personal computer is relatively cheap. A system with one physical shared memory is generally not scalable, as only a constant number of memory requests may be handled at one time, and other processors that want to access memory must wait their turn, effectively useless. The ultimate in scalable parallel systems is the network of personal computers, each completely self-sufficient, so that one can be used alone, or any number may be used together. Ideally, with a shared

[2] Ibid. p.19

memory system even different kinds of computers may be combined to run one program, transparently simulating one powerful parallel computer.

So when a theoretically "shared" memory location is updated, some processes, for example the ones running on the updating processor, will "know" about it before others, and unless the whole network waits for the message to propagate to the farthest node, some processes will be taking actions based on the memory location before the change, and some after. This has prompted the study of different *consistency levels* of memory, and which algorithms can be performed under each level, since a memory that has to live up to less strict standards of consistency spends less time waiting, and can work faster.

Specifically, Professor Abdelsalam Heddaya and Himanshu Sinha have made a formal classification of several of the most popular models of shared memory of differing levels of consistency.[3] Professor Heddaya and several graduate students have also implemented a system called *Mermera*, which allows a **C** programmer to use several different consistency levels of shared memory in a distributed program. It simulates a single shared memory by keeping a separate copy of all "shared" memory locations for each process, and passing messages between processes, usually on different computers, when a location is updated.

Models of Shared Memory, and Mermera

The most consistent model of computer memory is *static atomic* (SA),[4] when each memory access takes place at a specific, predictable time for all observers, between the invocation and return of the operation. So for example, if in a program one process continuously wrote the time since the program began to a memory location, and at some

---

[3] Abdelsalam Heddaya and Himanshu Sinha, "Coherence, Non-Coherence and Local Consistency in Distributed Shared Memory for Parallel Computing," BU-CS-92-004, Computer Science Department, Boston University.

[4] P.W. Hutto and M. Ahamad. "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories". In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems, Paris, France*, June 1990. Also available as GATech technical report GIT-ICS-89/39.

point in the program another process was to read that location, it would read the same time each time the program was run. A slightly weaker memory is called *dynamic atomic* (DA),[5] which allows an operation to take effect for all observers simultaneously, but at any point between its call and return. Therefore absolute time can no longer be kept. With this kind of memory running the same example program, the second processor might read a different value each time the program was run -- but, if it read that location twice consecutively, the second time value that it read would always be strictly *later* than the first time value that it read.

Since Mermera depends on a message passing protocol that does not guarantee a predictable time for another process to receive a passed message, Mermera cannot implement static atomic memory. Mermera could, theoretically, simulate dynamic atomic memory by sending a message from any process to all other processes after every operation, and not allowing two processes to perform operations at the same time, but this would be less efficient than even a sequential execution, all on one processor.

After the "atomic" memory levels, an operation can no longer be guaranteed to take effect between the time that it is called and the time that it returns. For the next strongest level of consistency, *externally consistent* (EC) memory,[6] an operation is only guaranteed to take effect for all observing processes in the same program some time before its result is ever used. For example, a cache with write-back gives this kind of memory: if a cache has a copy of a memory location and changes it, it might only invalidate the copy in the main memory, not change it. The main memory location will only be updated when another memory line has to enter the cache, or another processor wants to read the memory. An externally consistent parallel program will always return

---

[5] Ibid.
[6] Ibid.

the same result each time that it is run on the same inputs,[7] so there will not be any "race conditions" -- operations of two processes will always take place in the same, determinate, order. The only difference between an "atomically consistent" and an "externally but not atomically consistent" memory will be in the implementation at a lower level, not to the program, and likely not to the programmer. It is like two different answers to the koan about whether a tree falling without anyone to hear it makes a sound -- the "atomically consistent" answer would always be that it does, but an externally consistent answer could be either, since that would not affect any observers. For programmers, writing for any externally consistent memory is much the same. Any memory consistency below this one does not guarantee the same result from a program every time that it is run without special care.

Even weaker, and both the lowest boundary and the most commonly used definition for a *coherent* memory, is *sequential consistency* (SC).[8] The inventor, Lamport, describes SC as if the result of any execution of a program were the same as if the program were executed by one processor sequentially, with the operations of any one processor taking place in the sequence specified by its program. So when an operation takes effect it takes effect with relation to all processors, but an operation is only required to take effect before any operation *of the same process* uses its result. In one execution of the program, two operations of two processes could take place in a different order than in another. In an extreme case this can be simulated by running the program of a random processor completely when the last one finishes, without any communication except when a program finishes running. As an example, a multi-threaded single memory UNIX program is only sequentially consistent, in that its only guarantee to a process is that its

---

[7] In the example used for DA memory, the clock could be considered an input. Since even dynamic atomic operations can take an undetermined amount of time to complete, a different time will be "input" and read.

[8] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, c-28(9):241-8, September 1979.

nth statement will take effect before its n+1th, not that any number of other processes won't execute in the mean time. Programmers writing for a sequentially or less consistent system must already watch for race conditions.

Sequential consistency is achieved in Mermera with the CO- (coherent) write operation. When a process makes a COWrite, it sends a message that is guaranteed to reach all processes in the program, at which time the receiving processes update their copy of the relevant memory location. Even the writing process waits to receive the message before updating its local copy. All processes are guaranteed to receive all COWrite messages in the same order. This does not guarantee external consistency, since two processes could still make two COWrites at the same time, with one finishing before the other in one execution of the program but not another.

It is the specification that an operation must take effect with respect to all processors that makes any efficient execution of coherency or sequential consistency not scalable, because the entire network must wait for the processor that reacts the slowest -- and this slowest reaction time grows as the network diameter. Below the "coherent" memory boundary, the memory can no longer be simulated as being under the control of only one processor. The advantage of all less consistent kinds of memory is that a process doesn't have to simply wait for another processor to receive the result of every write operation, but can go on with a computation.

Heddaya and Sinha's paper gives a different formal definition of coherence, that a shared memory is coherent if all processes can agree on the order of any two writes to any one shared location. Later, I will show that this is not equivalent to, but is strictly weaker than sequential consistency. What is implemented in Mermera with the COWrite operations, however, is actually sequential consistency.

A non-coherent model, *causal* memory, was originally defined broadly, one "...that requires all processors to agree on the order of causally related effects (writes) but allows events not related by potential causality (*concurrent* events) to be observed in differing

orders."[9] This is a vague description that was tightened and formalized by Heddaya and Sinha in their classification paper, but not in a way that makes it easier to describe. Loosely, causal memory does not allow a read which observes two writes to return the value of the earlier one, if that read observed any events that perceived the writes in that order. So two events which disagree on the order of two writes cannot influence each other. A later event can observe both reads though. Mermera does not implement strictly causal memory, and there does not seem to be an easy way for it to do so.

The next weaker memory, *Pipelined Random Access Memory* (PRAM) is defined by the creator in terms of an implementation.[10] Every process keeps a copy of every shared memory location, and whenever it makes a change to a location, it broadcasts that fact, while going on with what it is doing. Whenever a process receives notice of a change, it updates the local copy. The effects of this are that one process might perceive two changes from different processes in one order, while another process in the same program might perceive the changes in the reverse order. The only guarantee is that two changes made by the *same* process will be perceived in the proper order by all other processes.

The implementation of PRAM in Mermera is slightly different from the described one. To save communication time, PRAM writes from each process are kept in a buffer, implemented immediately in the writing processes local copy, but only sent to other processes when the buffer fills, or when the process makes a COWrite. This way, fewer messages need to be sent to other processes, but the order of writes from each process is still preserved. PRAM and more coherent buffers are numbered consecutively by the sending process, so that a receiving process can implement them in the proper order, even if it should happen to receive them out of order.

---

[9] Hutto and Ahamad.
[10] R.J. Lipton and J.S. Sandberg. "PRAM: a Scalable Shared Memory." Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

Slow memory is even less constrained in that it only guarantees that two changes *to the same location* by the same process will be received in the proper order.[11] Therefore, a process might receive two writes to different locations by the same other process in reverse order. This gives the advantage to the implementation that it allows a change to be effectively "lost," that is allows for unreliable communication between processes, since it equivalent to a process whether it never gets notification of a change, or whether it gets it just before the very next write to the same location. Though slow memory and all less consistent memories allow occasionally faulty communication between processes, if communication were not at least sometimes successful, parallelism could not be really said to be implemented at all.

Mermera implements slow memory much as it does PRAM, making changes to the writing local copy of the shared memory immediately, but keeping changes to be broadcast to other processes in a buffer until the buffer fills. The only advantage that Mermera takes of slow memory is that slow buffers are numbered separately from reliable ones, so that a process is allowed to implement a buffer of updates even if it missed some slow buffers in the mean time. If a process receives a buffer late, that is a slow buffer that was sent before some buffer already implemented from the sending process, the tardy buffer would simply be ignored. However, the current protocol used for sending slow writes guarantees reception in the order sent, so the current implementation is effectively PRAM. The author hopes to be able to implement a less reliable protocol for Mermera shortly.

Next, Heddaya and Sinha define *local consistency* (LC) as only guaranteeing that memory accesses made by a specific process will seem to that process to take effect in the order it makes them. Absolutely no guarantees are made about memory accesses by other processes, though of course again, unless some inter-process communication is possible,

---

[11] Hutto and Ahamad.

this is not truly a shared memory. Heddaya and Sinha believe that local consistency "should be the weakest property that all shared memory systems must satisfy,"[12] but do not give examples of shared memory applications that do work with only local consistency.

Though strictly by the definition LC changes are allowed to be received by other processes, Mermera implements local consistency by not sending any messages to other processes about local writes at all. Therefore, using only local consistency for parallel programming is impossible on Mermera.

Finally, a *weak* memory[13] is one in which a read operation returns the result of any previous write. This does not seem very useful, as this means that even though a read returns one value now, a future read is not guaranteed to return the same value even if there were no intervening writes. This even violates the intuitive understanding of computer memory as a place to store data to be read later. Understandably, Mermera does not implement weak memory.

In their classification paper, Heddaya and Sinha prove that each type of non-coherent memory listed above is strictly weaker than the one above it, namely that any order of operations possible in a stronger memory is also allowable in a weaker one, but that the weaker memory also allows operation orders not allowable in the stronger one.

Other Shared Memory Models

Kai Hwang summarizes five types of shared memory consistency in his look at shared memory, and draws up a less formal hierarchy of his own. Out of the types he covers, sequential consistency has already been described, and *processor consistency*

[12] Ibid., p.16
[13] Hutto and Ahamad.

(PC), proposed by another author,[14] is equivalent to PRAM memory as described by Heddaya and Sinha, but the three other types deserve separate looks. None of them are yet simulated by Mermera, but all can be, with varying levels of modification of the Mermera code. They all have in common that they specify "synchronization" operations, besides read and write, that are used in places in the program to achieve the effect of a more consistent memory.

The first such is *Dubois, Scheurich, and Briggs* (DSB) *weak* memory.[15] This relies on "synchronization" commands in processes that are guaranteed to be sequentially consistent with respect to each other. When a process returns from a synchronization command, all of its previous synchronizations and memory accesses are guaranteed to be performed with respect to all other processes.

This memory model is easy to simulate using even the current Mermera, without any code changes. A DSB weak memory process would use PRAM writes for its writes, and when it needed to make a synchronization would issue a coherent write to any variable, probably a variable reserved for synchronization purposes, which would guarantee that all other processes will have received all its other writes when the synchronization is finished.

Similarly, this memory model is possible to classify roughly on Professor Heddaya's scale. It is strictly more consistent than slow memory, because writes are not allowed to be "lost" -- if a write were lost, and ever followed by a synchronization request, the write could never be performed with respect to the process that did not receive it and the synchronization could not complete. It can, however, be less consistent than PRAM, because two writes not separated by a synchronization command can be received in either

---

[14] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proceedings of the Third International Conference on Architecture,* pp. 422-431, 1988.

[15] M. Dubois, C. Scheurich, and F.A. Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442, 1986

order. In any case, this memory model is never more consistent than SC, sequential memory, since that is what issuing a synchronization command after every write achieves. There does not seem to be an easy way of simulating causal memory by using synchronizations between these extremes.

One problem with the terminology in this field is that everyone refers to less than coherent memories as "weak." Namely, Hwang's next model is known as *total store order* (TSO) *weak* consistency,[16] as opposed to DSB weak, or Hutto and Ahamad's simply weak.

This model gives each process a first-in-first-out buffer, a queue, to hold its writes to shared memory. Any reading by a process will first check its buffer for the most recent write which this process made to that location, and will return the value of that. If there are no writes to that location in the buffer, only then will the read check the global shared memory, which is guaranteed to be identical for all processes. This global shared memory is independently updated from the write buffers of all processes, removing the updating commands from the respective buffer as each write is made. This model also relies on a synchronization command, a *swap*, which indivisibly writes a value to a location and returns with the previous value of that location. A swap does not return until the issuing process's buffer has been emptied to global shared memory, guaranteeing that until the next write command that process will be reading from the global shared memory.

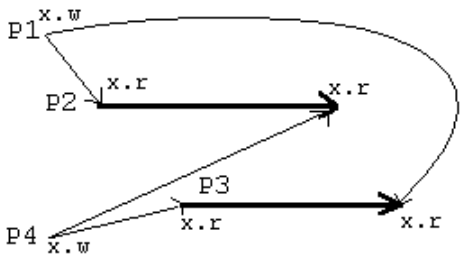
Issuing a swap after every write (or, making every write a swap, which will have the same effect) will make this model as consistent as sequential consistency, at the high end. At the low end, this model is clearly always at least as consistent as PRAM memory even without any swap commands, because global memory will be updated by writes only in the order that the writing process issued them. In fact, this model is even strictly more

---

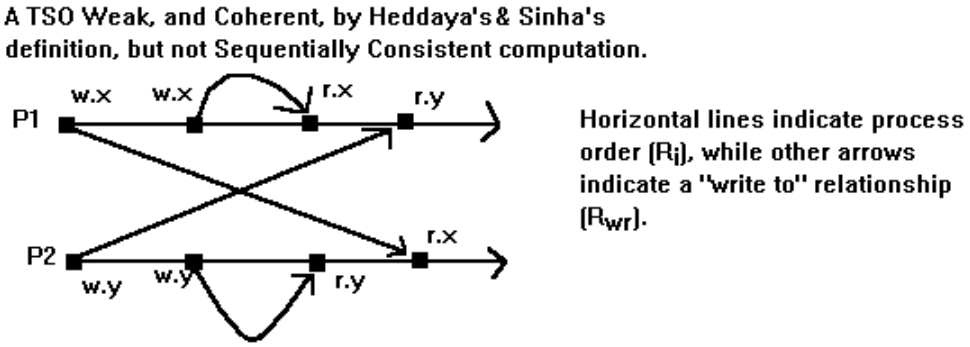
[16] P.S. Sindhu, J.M. Frailong, and M. Cekleov, "Formal Specification of Memory Modules," in Dubois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1992.

consistent than PRAM, because a value written by another process is available to all other processes at exactly the same time, the time when it updates global memory -- that is, there is a sequentially consistent global total ordering over all writes from one process read by another. It even seems to be stronger than causal memory, and as strong as coherent memory, by Heddaya and Sinha's definition, though this has yet to be formally proved. It would not, for instance, allow the computation given as an example of causal memory in Heddaya and Sinha's paper:

A computation that is causal but not TSO weak.



Since a process can only read a write from another process through the one global memory, either process 1's write would enter global memory before process 4's, in which case process 3 would not be able to read them in the opposite order, or process 4's write would enter global memory before that of process 1, in which case process 2 would not be able to read them in the opposite order. In fact, a TSO weak computation shows that sequential memory is strictly stronger, and not equivalent to Heddaya and Sinha's formal definition of coherent memory, by fitting the definition of coherent, but not that of sequential:



What is being pictured above is the following TSO weak computation: Process 1 writes two values to location x, which are put into its buffer. Then the global memory applies the first write from the buffer, but not yet the second. When process 1 reads from location x, it reads the value in its buffer, the second one written, while when process 2 reads from location x, it reads from global memory, reading the first value written by process 1. Process 2 writes to y similarly.

This computation is coherent by the definition, since all writes to location x are ordered, and all writes to location y are ordered -- no two processes read them in a different order. However, this computation is not sequentially consistent, since there is no possible total order over all the events: according to process 2's ordering, it must read x before process 1 makes the second write to x, therefore *process 2's second write must precede process 1's second write*. However, process 1's read of y must happen before process 2 makes the second write to y, so *process 1's second write must precede process 2's second write*, which is a contradiction.

Until further study of the subject, I suggest strengthening Heddaya and Sinha's formal definition of coherency to this: a computation is coherent if and only if its results are the same as if *all write operations of all processes* were executed in some sequential order, and the write operations of any single process appear in this sequence in the order specified by its program. Since the definition of sequential consistency requires *all* operations to be totally orderable in this way, this is clearly no stronger than sequential

consistency. Since read operations do not conflict with each other (any two reads without an intervening write will have the same result in either order), this is also at least as strong as sequential consistency, and therefore equivalent.

Practically, TSO weak memory is also possible to simulate with Mermera, though with a little more difficulty, requiring a slight addition to the internal Mermera code, which the author hopes to be able to do. All the writes of any process are buffered, like PRAM writes in Mermera, until either the buffer is full, or the user's process issues a swap command. Then the process sends the entire buffer as a coherent write, that is it waits for acknowledgment that all other processes received the update. Until that time, no changes are made to the writing processes own memory, instead, whenever the process wants to make a TSO read, it first searches its own write buffer for writes to that location, and returns the most recent one if there are any. A little savings in search time may be made by only keeping the most recent write to any location in the buffer -- it will still have the same effect when sent out.

The performance of the TSO addition to Mermera should be interesting to study, because it seems to offer some of the advantage of PRAM -- waiting for write updates to other processes in bunches, rather than one at a time -- while keeping a much more consistent global memory, even without any swaps. It will offer Mermera users a new memory consistency level, strictly between coherent and PRAM.

Finally, the most relaxed memory consistency described by Hwang is *release consistency*.[17] This relies on two different types of synchronization accesses, *acquires* and *releases* of specific shared memory locations. An acquire is a read operation that only returns when the process has gained permission to access the relevant shared memory,

---

[17] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990. This is not the same as the Release Consistency mentioned in passing in Heddaya and Sinha's classification paper. Again, terminology in this area leaves something to be desired.

while a release is a write operation that gives away this permission. These synchronization accesses are process consistent (PRAM consistent) with each other.

This might be better understood by considering a local cache implementation. An acquire is a read by a process from shared memory into a local cache, and a release is a write of the value from that cache back to shared memory. In this way, release consistency is reminiscent of TSO weak memory, in that each process reads from and writes to a local buffer. However, unlike TSO weak, even the shared memory is allowed to be inconsistent between processes, as long as it is processor consistent. This memory has a similar relationship to PRAM memory as TSO weak memory has to coherent memory -- when a release operation is performed an update message is sent out to other processes, but the writing process does not wait for all other processes to receive the message, but just goes on with its work, updating its copy of the shared memory in reaction to other processes' messages as it receives them. It provides a savings over PRAM memory in that update messages do not have to be sent for all writes to memory, but only when the process decides to "release" them -- a process might decide to acquire several variables, write and read to them intensively over a period that would cause many PRAM messages to be sent, and only release them when it no longer needs them, causing only one update per variable.

Again, since release messages are guaranteed to be received in the order that the sending process sent them, release consistency is strictly stronger than slow memory. However it is no stronger than PRAM, or process consistent memory, since that is what results if every write is immediately followed with a release. So release consistency fits reasonably neatly between two points on the chart.

Notice that since only process consistency of synchronization commands is guaranteed, permission to acquire a variable is always granted immediately -- it is impossible to guarantee that process 1 will not acquire a variable at the same external time that process 2 has already acquired it, since process 2 did not wait for process 1 to receive

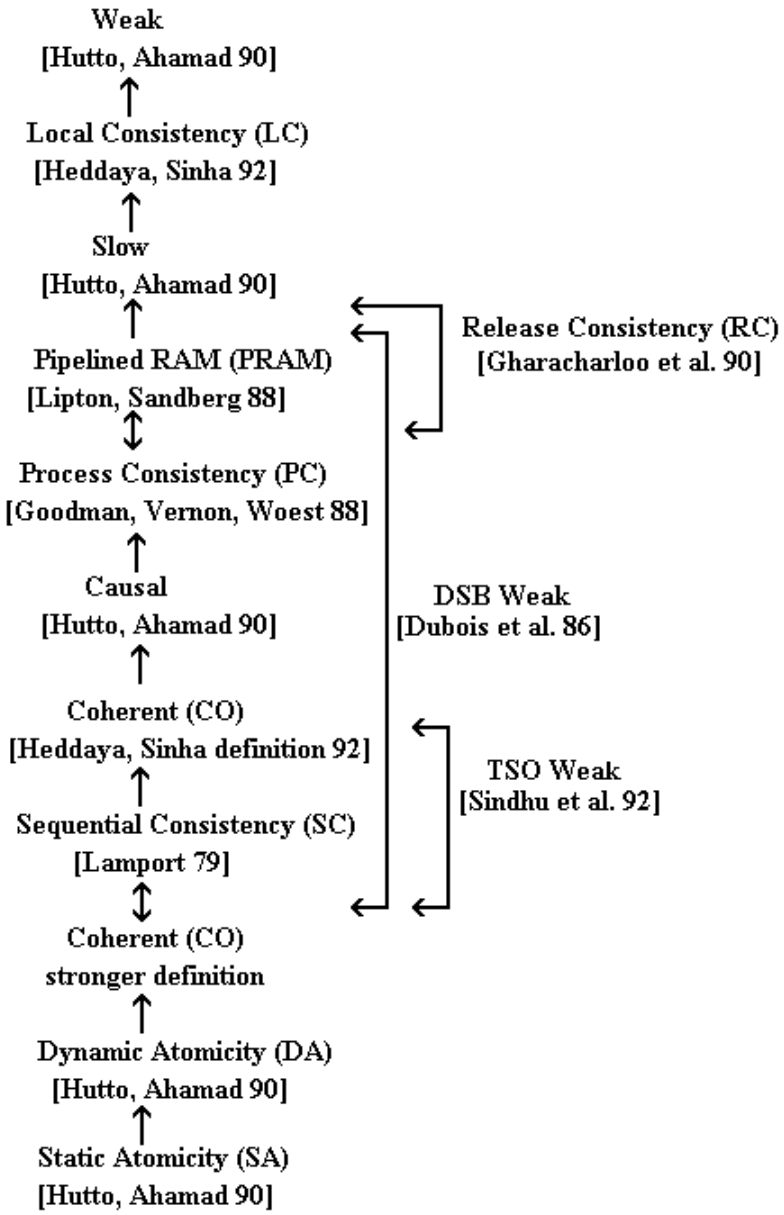
process 2's acquire message (waiting would guarantee sequential consistency of acquire commands). So acquire commands need not be broadcast to other processes at all.

The Mermera implementation of release consistency would require a moderate amount of work within the Mermera internals. Mermera would keep a separate buffer for shared memory locations acquired and not yet released by the process. Any reads and writes the process made would go to that buffer. A release would be treated as a PRAM write, that is it would be buffered to be sent to other processes and immediately implemented locally without waiting for confirmation.

Some computations might even get more of a benefit from using RC than even Slow memory, because RC would reduce the number of communications between processes. An RC process could acquire a memory location, and make countless changes to it, while only notifying other processes about the last change, at release time. In addition to which, RC still guarantees reliable communication between processes.

All of these synchronization operation dependent models require the programmer to spend more effort considering shared memory requirements in writing code, which to some extent defeats the purpose of using any shared memory model. Only the TSO weak model adds anything new to the list of shared memory models without the synchronization aspect. But the effort of the programmer considering synchronizations seems to be still less than that of writing for a raw message-passing model, as evidenced by the amount of effort spent inventing synchronization-dependent shared memories.

A relative ordering of shared memory models



Summary

In this paper, I summarized many models of shared memory of differing levels of consistency that were formally defined by Professor Abdelsalam Heddaya and Himanshu Sinha. I also described the implementations of the four models that were implemented in

Heddaya and Sinha's Mermera system for distributed computing. In addition, I described three additional models of shared memory mentioned in Kai Hwang's book *Advanced Computer Architecture*, classified them in relation to Heddaya and Sinha's classification, and showed how they could be implemented in Mermera, with different degrees of modification of the Mermera internal code.

Future Work

I am working on a modification of Mermera to use an unreliable message passing protocol, to more correctly implement Slow memory, and speed up the fastest shared memory available. This modification will also manage the shared memory message passing through one process, and the use of signals, rather than forking a separate process to handle messaging, which should speed up the entire system, though this will be invisible to the programmer -- the external results should be unchanged.

Finally, I intend to formally define the three new memory models I described in this paper using Heddaya and Sinha's formalisms, and implement them in Mermera, much as I have shown above. This will enable testing of various models against each other to see how much of a relative speedup each provides, and whether it is worth the overhead in programmer effort of worrying about making synchronization calls.