Representation of coherency classes for parallel systems

Representation of Coherency Classes for Parallel Systems

Walter Hussak
Department of Computer Studies,
University of Technology,
Loughborough, UK.

John A. Keane
Centre for Novel Computing,
Department of Computer Science,
The University, Manchester, UK.

Abstract

Some parallel applications do not require a precise imitation of the behaviour of the physically shared memory programming model. Consequently, certain parallel machine architectures have elected to emphasise different required coherency properties because of possible efficiency gains. This has led to various definitions of models of store coherency. These definitions have not been amenable to detailed analysis and, consequently, inconsistencies have resulted.

In this paper a unified framework is proposed in which different models of store coherency are developed systematically by progressively relaxing the constraints that they have to satisfy. A demonstration is given of how formal reasoning can be carried out to compare different models. Some real-life systems are considered and a definition of a version of weak coherency is found to be incomplete.

1 Introduction

It has long been realised that physically shared memory parallel computers have problems of scalability beyond a limit of 30-50 processors [2]. This problem arises because the memory is accessed by a communications network, usually a bus, the capacity of which becomes overloaded with increasing numbers of processors. However, the programming model of such machines is very convenient for programmers: asynchronous processes communicate and synchronise via variables residing in the shared memory. Memory access and location is transparent to the programmer.

Physically distributed memory machines, that contain closely-coupled processor-store pairs, offer potentially unlimited scalability because communication bandwith scales with extra processors. However, such machines enforced a distributed memory programming model where asynchronous processes communicate and synchronise via message-passing. Communication is via the content of a message and synchronisation occurs, in that a message must be sent *before* it is received. The programming model requires knowledge of memory location in the system and is more complicated than the shared memory programming model.

In recent years, parallel computer architects have attempted to provide the convenience of the shared memory programming model with the scalability of physically distributed memory machines.

To provide *precisely* the same programming model that physically shared memory machines provide, on a physically distributed memory machine, requires extensive support in either hardware or software. If provided in hardware this usually means that the price/performance of the machine increases because the hardware is necessarily more sophisticated. If implemented in software there is necessarily much more co-ordination at operating system level, and thus between processor-store pairs and a consequent loss in overall performance. In both cases, the requirement on the communication network increases significantly if a shared memory model is provided.

It has become apparent that some applications do not require a precise imitation of the behaviour of the physically shared memory programming model. Consequently, partly because of the application requirements and partly because of *perceived* efficiency gains, various parallel machine architectures have elected to emphasis different required *coherency properties* of shared memory. So far, such required properties and efficiency gains have been based on intuitive architectural tradeoffs and application areas targeted for the architectures. Such intuitions have been documented in an ad hoc fashion and have not permitted detailed analysis and comparison between different models. This paper aims to elevate the study of coherency classes of parallel and distributed systems to the level that serializability has been studied in database systems.

1063-6374/93 $03.00 © 1993 IEEE

A unified formal framework is given in which different models of store coherency are developed systematically by progressively relaxing the constraints that the models have to satisfy. Three models of store coherency, that have been found to be useful by parallel machine designers, are defined: *strong coherency, sequential consistency* and *weak coherency.* The framework differs from existing shared memory multiprocessor models, as for example in [1] and [13], in that it also allows for the representation of coherency classes for distributed memory multiprocessor systems.

The formal framework is in section 2. Section 3 presents store coherency in this framework, and section 4 sequential consistency. A definition of a weak coherency class is given in section 5 together with a demonstration of how formal reasoning is performed between different coherency classes. In section 6, the coherency properties of some real-life systems are considered and a definition of weak coherency is shown to be incomplete. Conclusions are drawn in section 7.

2 Formal Framework

The following abstracts the essential features for developing and communicating ideas about coherency classes. A *run* in a system is a 4-tuple $(\mathcal{X}, \mathcal{P}, \ll, sees)$, where

- $\mathcal{X}$ is a set of variables or storage locations X, Y, ...
- $\mathcal{P}$ is a finite set of processes where each process is a sequence of events each of which is either a read or a write to one of these variables
- $\ll$ is a total order on the set of all the events in all the processes, represented some absolute physical temporal ordering
- if $\mathcal{RE}$ and $\mathcal{WE}$ are the set of all read and write events in all the processes, *sees* is a function

$$sees : \mathcal{RE} \rightarrow \mathcal{WE},$$

which indicates which previous write a read event "sees". This function has three basic properties, given below, to make the definition sensible.

Notes

1. Events are simply reads or writes to variables. Any storage system can be represented by the "fetches" and "stores" to some variable or storage location set that occur.

2. A read event to a variable X will be denoted

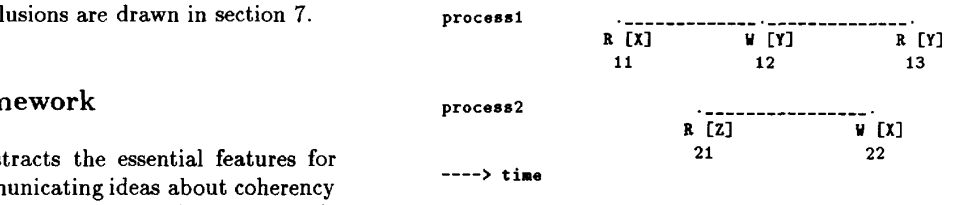
$$R[X]$$

and a write event

$$W[X]$$

This is the notation used in serializability theory [12].

3. Events in a process i will be subscripted by i and further subscripted by consecutive numbers. The run will be represented by a 'space-time' diagram as in [Lam78], though with 'time' represented horizontally and 'space' vertically, and the definition of the *sees* function. For example,



has $R_{11}[X] \ll R_{21}[Z]$. For process i the (irreflexive) total order of the sequence of events will be denoted $<_i$.

4. The temporal ordering $\ll$ provides an external context or physical clock as in [9]. Here, it is needed to distinguish certain coherency classes.

5. The read events correspond to reads being received and the write events correspond to writes being issued. The scenario of 'actual' writes taking place in a different order to that in which they were issued, as for example in the PSO model of [13], can be accommodated by an appropriate choice of the *sees* function.

6. A *system* will just be a collection of runs. This follows the common practice in concurrency theory of describing a system as a set of possible runs (or traces).

sees function

The three conditions needed so that the *sees* function represents a meaningful storage system are:

- **Variable consistency**

  A read of a variable X can only see a write to that same variable

  $$sees(R_{ij}[X]) = W_{kl}[Y] \Rightarrow X = Y$$

- **Temporal consistency**

  The value of a variable can only be read after it has been physically written

  $$sees(R_{ij}[X]) = W_{kl}[X] \Rightarrow W_{kl}[X] \ll R_{ij}[X]$$

- **Local consistency**

  If a read event in a process sees a write event that occurred in that process, it must be the last such write event.

  $$\neg((sees(R_{ij}[X]) =$$

  $$W_{ik}[X]) \wedge (W_{ik}[X] <_i W_{il}[X] <_i R_{ij}[X]))$$

The different classes of coherency will be characterised by an appropriate choice for the *sees* function.

3 Strong Coherency

A *strongly coherent* store model corresponds most closely to the behaviour of physically shared store parallel machines. When a variable is to be written to by a process, a *write* lock is obtained thus excluding any other processes from reading that variable until the write lock is given up. It is clear that a strongly coherent system is one in which a write becomes visible to every process as soon as it occurs. Thus, any read in any process always sees what is, temporally, the latest write.

Definition

A *strongly coherent run* is one in which if

$$W_{kl}[X] \ll R_{ij}[X]$$

and

$$(W_{kl}[X] \ll W_{gh}[X] \ll R_{ij}[X]) \Rightarrow (k = g \wedge h = l)$$

then

$$sees(R_{ij}[X]) = W_{kl}[X]$$

A *strongly coherent system* is a set of strongly coherent runs.

The strongly coherent model arises from the *physical* behaviour of physically shared store parallel machines, i.e. a location in the physically shared memory can only be accessed by one processor at a time, thus there is the notion of atomic update of a variable. Since there is only one copy of the variable in the system, any subsequent reads *must* see the new value written by the most recent write[1].

Since this is the model supported by shared memory machines, this class of coherency is seen as being the *most obvious* when shared memory is provided on a physically distributed memory machine.

4 Sequential Consistency

Lamport [10] defines a sequentially consistent system:

> A system is sequentially consistent if the result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

The sequentially consistent model gives a statement about the *logical* behaviour rather than any physical behaviour, i.e. the logical behaviour that a programmer might expect from a multiprocess program.

Informally, a *sequentially consistent* run is one which has the "same effect" as a single process running the events of the constituent processes in some sequential order preserving the order of events in all the processes, though not necessarily the overall global temporal order of events. Before a formal definition can be given the words "same effect" need to be made precise.

4.1 τ - equivalence

Formally, two sequentially consistent runs have the "same effect" if they are equivalent in the following (rather technical) sense.

Two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ are $\tau - equivalent$ if:

1. $\mathcal{R}_1$ and $\mathcal{R}_2$ have the same processes with the same sequence of reads and writes.

[1] In many such systems each processor has a cache but snoopy caches enforce the atomic update.

2. Given:

   (a) Any value domain for the variables X, Y, ...,

   (b) Any set of initial values for the variables from the domains,

   (c) Every value written to a variable by a process in a write event is an uninterpreted function of previous values read by that process

   then, for all processes, $\mathcal{R}_1$ and $\mathcal{R}_2$ read the same sequence of values.

Put simply, the sequence of values read by a given process are the same in $\mathcal{R}_1$ and $\mathcal{R}_2$[2]. The formal definition of sequential consistency is as follows.

4.2 Definition of Sequential Consistency

A run $\mathcal{R}$ is *sequentially consistent* if it is τ-equivalent to a strongly coherent run. A *sequentially consistent system* is a set of sequentially consistent runs. The following theorem gives a more amenable check for sequential consistency.

4.3 Theorem

$\mathcal{R}$ is sequentially consistent if (and only if) there is an irreflexive total order $<$ on all the events of all the processes in $\mathcal{R}$ such that:

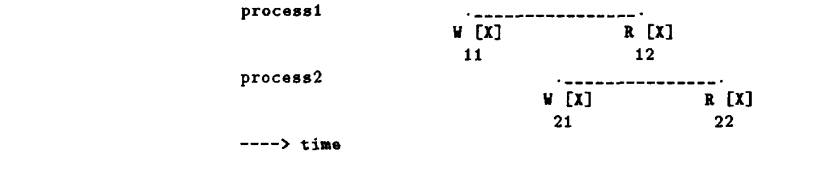
(i) $event1 <_i event2 \Rightarrow event1 < event2$,

(ii) $sees(R_{ij}[X]) = W_{kl}[X] \Rightarrow W_{kl}[X] < R_{ij}[X]$,

(iii) $W_{kl}[X] < W_{gh}[X] < R_{ij}[X] \Rightarrow sees(R_{ij}[X]) \neq W_{kl}[X]$.

4.4 Example

The following is an example of a sequentially consistent system that is not strongly coherent

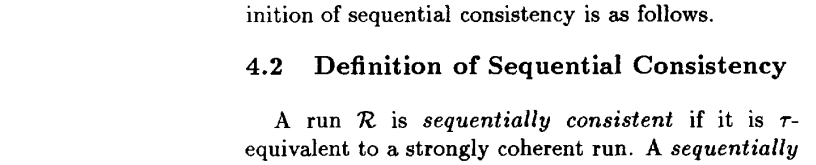


$sees(R_{12}[X]) = W_{11}[X]$, $sees(R_{22}[X]) = W_{21}[X]$

[2]The definition is related to τ – *serializability* from serializability theory, and a run $\mathcal{R}$ with finitely many events is τ-serializable in the sense of [15] if it is τ-equivalent to a run where $\mathcal{R}$'s processes have been placed 'end-to-end' in some serial order. The notion of τ-equivalence is a type of observational equivalence as in concurrency theory [11].

5 Weak Coherency

A weakly coherent system is one in which there is defined a 'happens before' or 'causally affects' relation between events in the system, independently of the temporal order $\ll$ of physical time. Typically, this would be as in [9] where a 'happens before' relation is defined on the set of events of a system where message passing is taking place and where the point at which the message is sent is deemed to 'happen before' the point at which the message is received. Such a system is represented in a space-time diagram with arrows representing messages.



5.1 Definition

A run $\mathcal{R}$ is *weakly coherent* if there is an irreflexive partial order $\rightarrow$ on the events of $\mathcal{R}$ such that:

(i) $event1 <_i event2 \Rightarrow event1 \rightarrow event2$,

(ii) $R_{ij}[X] \rightarrow W_{kl}[X] \Rightarrow sees(R_{ij}[X]) \neq W_{kl}[X]$,

(iii) $W_{kl}[X] \rightarrow W_{gh}[X] \rightarrow R_{ij}[X] \Rightarrow sees(R_{ij}[X]) \neq W_{kl}[X]$

(iv) $(sees(R_{fg}[X]) = W_{lm}[X]) \wedge (sees(R_{hi}[X]) \neq W_{lm}[X]) \wedge (R_{fg}[X] \rightarrow R_{hi}[X] \rightarrow R_{jk}[X]) \Rightarrow (sees(R_{jk}[X]) \neq W_{lm}[X])$

A *weakly coherent system* is a set of weakly coherent runs.

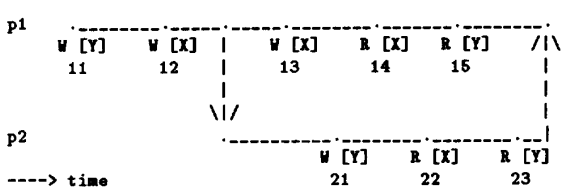
The condition (iii) is an extension of the local consistency condition on the *sees* function. It states that if a write 'happens before' a read, then the read cannot possibly see an earlier write. Condition (iv) is a consequence of 4.3 (i), (ii) and (iii) when dealing with sequentially consistent systems. In general, it is possible to have runs satisfying conditions (i), (ii) and (iii) but not (iv). The idea of (iv) is that once a write has been seen, and a subsequent read sees a different write, then that previous write is an 'old' value and should not be seen again.

It is easy to show that a sequentially consistent system is weakly coherent. The converse does not hold, so that weak coherency, logically, is a different notion. The advantage of having a formal framework is that

such supposed logical distinctions, arising from architectural intuitions, can be proven. Proofs of equivalences of systems can also be provided. As an illustration, the following example gives a weakly coherent system that is not sequentially consistent.

5.2 Example

Consider the following system $\mathcal{S}$



$$sees(R_{22}[X]) = W_{12}[X] \quad (1)$$

$$sees(R_{23}[Y]) = W_{21}[Y] \quad (2)$$

$$sees(R_{14}[X]) = W_{13}[X] \quad (3)$$

$$sees(R_{15}[Y]) = W_{11}[Y] \quad (4)$$

The check that this is a weakly coherent system is straightforward and is not given here.

Theorem

The system $\mathcal{S}$ is not sequentially consistent.

Proof

Assume, on the contrary, that $\mathcal{S}$ is sequentially consistent. Choose an irreflexive total order $<$ satisfying conditions (i), (ii) and (iii) of 4.3. Now, if

$$W_{13}[X] < R_{22}[X], \quad (5)$$

then as

$$W_{12}[X] <_1 W_{13}[X], \quad (6)$$

by 4.3(i),

$$W_{12}[X] < W_{13}[X], \quad (7)$$

and therefore

$$W_{12}[X] < W_{13}[X] < R_{22}[X] \quad (8)$$

By 4.3(iii),

$$sees(R_{22}[X]) \neq W_{12}[X] \quad (9)$$

This contradicts (1) and therefore (5) cannot be true. Thus,

$$R_{22}[X] < W_{13}[X] \quad (10)$$

Also,

$$W_{11}[Y] < W_{21}[Y] < R_{15}[Y] \quad (11)$$

cannot be the case as then $R_{15}[Y]$ would be required to see $W_{21}[Y]$ which contradicts (4). So, either

$$W_{21}[Y] < W_{11}[Y] \quad (12)$$

or

$$R_{15}[Y] < W_{21}[Y] \quad (13)$$

must hold.
Suppose (12) holds. Then,

$$R_{23}[Y] < W_{11}[Y] \quad (14)$$

else

$$W_{21}[Y] < W_{11}[Y] < R_{23}[Y] \quad (15)$$

and 4.3(iii) would mean that $sees(R_{23}[Y]) \neq W_{21}[Y]$ contradicting (2). Now,

$$R_{22}[X] <_2 R_{23}[Y] \quad (16)$$

Thus,

$$R_{22}[X] < R_{23}[Y] < W_{11}[Y] \text{ by(13) } < W_{12}[X] \quad (17)$$

By 4.3(ii), this means that

$$sees(R_{22}[X]) \neq W_{12}[X] \quad (18)$$

This contradicts (1) and so (12) cannot hold.
Finally, suppose (13) holds. then,

$$W_{13}[X] <_1 R_{15}[Y] < W_{21}[Y] <_2 R_{22}[X] \quad (19)$$

Therefore, $W_{13}[X] < R_{22}[X]$ contradicting (10). Thus, the original assumption that $\mathcal{S}$ is sequentially consistent is untenable. □

5.3 Remarks

The weakly coherency model appears to be particularly useful in parallel symbolic applications. This appears to be because such systems tend to have side-effect free computational models and thus allow implementations that do not require a totally consistent view of the underlying store at all times. The weakly coherent model reduces the amount of synchronisation required by a strongly coherent model, at the expense of making language run-time subsystems define and implement their own coherency models, and thus be slightly more complex.

Strong coherency necessarily involves more network traffic, if an application does not require it then it is an expensive redundant feature.

6 Case Studies

In the following sections a parallel system that provides a strongly coherent model, the KSR1, and a parallel system that provides a weakly coherent model, EDS, are discussed.

6.1 KSR

The KSR1 parallel system [4] was designed as a general-purpose machine. The KSR memory system provides a store model where all writes to a shared variable are immediately seen by any subsequent read of that variable in any process.

The KSR model thus provides the model of strong store coherency defined in section 3. As has been pointed out, strong coherency provides the programmer with the logical behaviour expected, i.e. sequentially consistent behaviour.

Essentially the KSR approach has been to design hardware that can implement strong store coherency. The KSR physically distributed memory system imitates a physically shared memory system.

Because strong coherency is provided for in the hardware of the KSR, it does not appear possible for the system to provide a weakly coherent, *lazy invalidation* store model[3].

Performance Considerations

The KSR is a non-uniform memory access (NUMA) time system. Each set of 32 processors are contained in a *search engine:0*. A search engine:0 provides the hardware that enables strong store coherency to be achieved across the 32 processors. In turn, a *search engine:1* provides the hardware that enables strong store coherency between each 32-processor set, i.e. across an entire system.

On every write to a shared variable an invalidate message must be sent to ensure that all copies of the variable are made invalid throughout the system. When this invalidation has been carried out a processor can write to the variable. Potentially this can cause a large amount of network traffic across a number of search engines — the maximum configuration has 34 search engines. In the worst case, it is possible to imagine the system performance being bound on the frequency of system-wide invalidation messages. KSR acknowledge this possibility but suggest that locality within a thread and between threads on the same search engine:0 should ensure that in the vast majority of cases there is no system-wide network traffic for invalidation broadcasts.

A price/performance issue arises if it can be shown that an application only requires a weakly coherent store model then the highly sophisticated KSR hardware design becomes redundant. In addition, it is possible that such applications will not perform as well as they might without the extra hardware sophistication, as strong coherency will cause unnecessary invalidation messages to pass through the system.

[3] In [5] a weakly coherent model is built, in software, on top of a strongly coherent model.

6.2 EDS

It is recognised that to provide strongly coherent store in software is 2 to 3 orders of magnitude slower than in hardware [4]. Hence, parallel systems architects' have recognised that certain applications do not *necessarily* require this type of coherence and that a weaker model will suffice. The advantage of this weaker model is that it increases performance by reducing network traffic and software co-ordination.

The EDS parallel system [14] was designed as a *declarative* system. Its aim is to run relational databases systems, functional languages and logic languages efficiently. EDS has, partly because of the *declarative system* emphasis, focussed upon identifying applications that require only a weakly coherent model, and efficiently supporting the model. A definition of a weaker form of coherency was produced for EDS.

The EDS definition underwent several iterations from the first version in [8] to a final version in [3]. The final version is given below. A *thread* is just a process and a CES [4] can be assumed to be a message from one thread to another. The terms 'see', 'can see' and 'visible' were presented informally.

A partial ordering of events, denoted by $<<$, is defined as follows:

1. If A and B are actions in the same thread and A comes before B, then $A << B$.
2. If A is the initiation of a CES by one thread and B is the corresponding point of synchronisation in another (potentially waiting) thread, then $A << B$.
3. If $A << B$ and $B << C$ then $A << C$.

Two events A and B are said to be *concurrent*, denoted by

$$A \parallel B,$$

if

$$\neg\, A << B \text{ and } \neg\, B << A$$

4 If $W << R$, then W must be *visible* to R. Visible means that R must read the data that have been written at W, if there is no intermediate or concurrent access W', i.e., if there is no W' with $W << W' << R$ or $W \parallel W' << R$.

[4] *Coherency Establishing Synchronisation.*

5 If $R << W$, then R must not read the data that are written at W.

6 Concurrent writer and reader: If $W \parallel R$, then it is undefined whether or not the reader will see the writers modification.

7 Concurrent writers: If $W' \parallel W << R$, then

(a) $R << W'$ is impossible ($W << R << W'$ implies $W << W'$).

(b) If $W' \parallel R$ then (6) applies and it is undefined whether R will read the value written by W or by W'. It will read either of those values.

(c) If $W' << R$, then it is undefined whether R will read the value written by W or by W'. It will read either of those values.

According to the definition, the following is permitted: Thread $th1$ performs a write W_1, after which it issues a CES to thread $th3$, such that $W_1 << R_1$ (see the definition of $th3$ below). Thread $th2$ performs a write W_2, after which it issues a CES to thread $th3$, such that $W_2 << R_1$. Thread $th3$ performs a succession of reads $R_1 << R_2 << R_3 \ldots$. Then, $W_1 << R_1$, $W_2 << R_1$ and $W_1 \parallel W_2$ (assuming that $th1$ and $th2$ do not communicate), and so R_1 can see W_1 by 7(c). Similarly, $W_1 << R_2$, $W_2 << R_2$ and $W_1 \parallel W_2$, and so R_2 can see W_2 by 7(c). It is possible to have R_1 see W_1, R_2 see W_2, R_3 see W_1, R_4 see W_2, ...

This is clearly not an intended behaviour of the system and should be disallowed.

At this point the benefits of using a formal framework should be mentioned. Firstly, formal reasoning is facilitated as was demonstrated in 5.2. Secondly, the actual process of formal specification forces consideration, at an early stage, of many features that might escape attention in an informal specification. In the case of the EDS weak coherency requirements, consideration in the context of the formal framework gives that condition (iv) of 5.1 is a requirement. Its inclusion disallows the pathological behaviour described. Without the requirement, no sensible detailed reasoning about the system would be possible. Use of the formal framework also permits needless iterations of requirements being produced due to inaccuracy of expression.

The EDS model, with the necessary addition, provides a weakly coherent store model. The EDS coarse-grained Parallel Lisp system [6] makes use of the weakly coherent store model as does the EDS relational database query distribution mechanism [7].

7 Conclusions

The architectural design of parallel systems is a set of engineering tradeoffs that emphasise features that the architects consider paramount. The intuitions behind such emphases requires a formal framework to enable decisions to be evaluated prior to the extensive and expensive prototyping phase of design.

This paper contributes to this aim by providing a formal framework in which different store coherency classes can be represented, and thus such architectural issues can be considered on a quantitative rather than qualitative basis. The formal framework has been used for a variety of different coherency models, one based on physical properties, one on certain observational properties and one applicable to a general class of message-passing systems. Some real-life systems have been analysed and the incompleteness of a definition of weak coherency has been identified and corrected.

With the advent of parallel applications specifying minimal coherency requirements for their implementation, the issue of comparing models of coherency of different parallel systems is an important one from the point of view of porting such applications. Questions of stronger or weaker forms of coherency can only be resolved satisfactorily if some sort of reasoning is possible. This paper provides a demonstration of how this might be carried out.

Acknowledgements

The authors wish to thank all their former colleagues in the Esprit EDS project for many helpful discussions. Walter Hussak wishes to acknowledge Lothar Borrmann of Siemens for discussions on formal models for the EDS store coherency requirements. John Keane wishes to acknowledge all members of the CNC and CGU at the University of Manchester.

References

[1] S.V. Adve and M. D. Hill, Weak Ordering - A new definition, *Proceedings of 17th International Symposium on Computer Architecture*, 1990.

[2] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Co., 1989.

[3] L. Borrmann and M. Herdiecekerhoff, A Coherency Model for Virtual Shared Memory, *Proceedings of Int. Conf. on Parallel Processing*, St. Charles, Illinois, August 1990.

[4] S. Frank, H. Burkhardt and J. Rothnie, The KSR1: Bridging the Gap Between Shared Memory and MPPs, *Proceedings of Compcon'93*, pp. 285-294, San Francisco, CA, February 1993.

[5] W.K. Giloi, C. Hastedt, F. Schoem and W. Schroeder-Priekschat, A Distributed Implementation of Shared Virtual Memory with Strong and Weak Coherence, pp. 23-30, in *Distributed Memory Computing*, A. Bode (Ed.), LNCS-487, Springer-Verlag, 1991.

[6] C. Hammer and T. Henties, Using a Weak Coherency Model for a Parallel Lisp, pp. 32-41, in *Distributed Memory Computing*, A. Bode (Ed.), LNCS-487, Springer-Verlag, 1991.

[7] N. Holt, Virtual Shared Memory in Commercial Applications, Virtual Shared Memory Symposium, University of Manchester, September 1992.

[8] P. Istavrinos, The Process Control Language, EDS.DD.1S.0007, EDS Project Document, 1989.

[9] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Communication ACM*, 21, pp.558-565, 1978.

[10] L. Lamport, How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers*, C-28 (9), pp. 690-691, 1979.

[11] R. Milner, *Communication and Concurrency*, Prentice-Hall International, 1989.

[12] C. Papadimitriou, The Serializability of Concurrent Database Updates, *Journal of ACM*, 26 (4), pp. 631-653, 1979.

[13] P.S. Sindhu, J-M. Frailong and M. Cekleov, Formal Specification of Memory Models, Palo Alto Research Center, Technical Report CSL-91-11, December 1991.

[14] C.J. Skelton, C. Hammer, M. Lopez, M. *et al.*, EDS: A Parallel Computer System for Advanced Information Processing, in *PARLE'92*, (D. Etiemble and J.-C. Syre Eds.), pp. 3-18, LNCS-605, Springer-Verlag.

[15] K. Vidyasankar, Generalized Theory of Serializability, *Acta Informatica*, 24, pp. 105-119, 1987.