Supporting Ada 95 Passive Partitions in a Distributed Environment

Frank Mueller

Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de *phone: (+49) (30) 20181-276*

Abstract. Ada 95 passive partitions, containing passive library units, provide the means to distribute data within a network of workstations. This paper shows how passive partitions can be implemented *via* distributed shared virtual memory (DSM). DSM provides the logical view of a portion of memory shared between physically distributed workstations within a network. In this paper, we relate design issues and operational characteristics of DSM systems to the semantics of shared passive library units as specified in the Ada 95 distributed system annex. We designed and implemented such a DSM system, operating at the granularity of pages, in a portable fashion over POSIX threads. The DSM system establishes its own communication services and is completely independent of any Ada runtime system support. Protected objects are supported *via* a novel prioritized protocol for distributed mutual exclusion. We integrated the DSM system with the Gnu Ada Translator (Gnat) and its environment to support active partitions. The DSM system integration required minimal changes to the Gnat environment and remains completely transparent to the user and the Ada runtime system. This is the first implementation of shared passive library units within an Ada 95 compilation environment, to our knowledge. The DSM support can provide considerable performance advantages over alternative approaches with repeated remote accesses, since with DSM only the first access to remote data results in communication overhead while consecutive accesses may take the same time as local memory accesses. Using the DSM paradigm, distributed applications can be designed to access distributed shared data efficiently. Furthermore, the shared memory model inherent to the Ada language can be used without modifications in a distributed environment.

1 Introduction

Distributed systems are gaining in popularity because they provide a cost-effective means to exploit the existing processing power of networks of workstations. Nonetheless, there are a number of problems associated with the transition to distributed systems, which often limit the performance gains.

One problem is caused by the communication medium between processing nodes (workstations), which often creates a bottleneck. Recent advances increase the throughput of communications by at least an order of magnitude (*e.g.*, FDDI, FastEther, ATM), which improves the situation to some extent. However, access to data on remote nodes will always be slower than access to local data. Thus, limiting the number of remote accesses will always be a design goal of distributed systems.

Another problem is rooted in the absence of a global state and the absence of an easily understood programming model to address this issue. As a result, distributed algorithms are often hard to understand and require explicit communication directives (*e.g.* message passing). Simple solutions often employ a centralized approach, where a single server provides a global state to its clients, but the server may create a bottleneck again.

An alternative solution is provided by *distributed virtual shared memory (DSM)* [21, 22]. DSM provides the logical view of a portion of an address space, which is shared between physically distributed processing nodes, *i.e.* a global state. Address references can be distinguished between local memory accesses and DSM accesses, *i.e.* an architecture with non-uniform memory access (NUMA) is created. However, the well-understood paradigm of concurrent programming can be used without any changes, since the communication between nodes is transparent to the programmer.

The distributed systems annex of Ada 95 [8] supports the abstraction of global data shared between processing nodes *via shared passive library units* associated with a certain node (called a *passive partition*). Passive partitions can be supported in various way. For example, message passing or remote procedure calls (RPC) could be employed to fetch data of passive libraries from a certain node each time it is accessed. A much more efficient alternative is provided by implementing passive partitions *via* DSM. Here, only the first data access results in communication overhead to obtain ownership of the data. While a node owns the data, consecutive accesses are as fast as local memory accesses. This paper describes how the Ada 95 model of passive partitions can be mapped onto the DSM model.

2 The DSM Model

DSM can be implemented in hardware or in software. This paper focuses on the latter since it is aimed at existing networks of workstations. A major distinction between different DSM systems is given by the granularity of shared data. Sharing can be accomplished at the word level, the page level, at the scope of objects or within tuples. This paper focuses on page sharing because accesses can be monitored at this level *via* operating system and hardware support available in most systems. Figure 1 depicts such a system with a varying number of CPUs per node (single processor or shared-memory multi-processors) connected by an arbitrary communication medium. A portion of each node's addressing space is

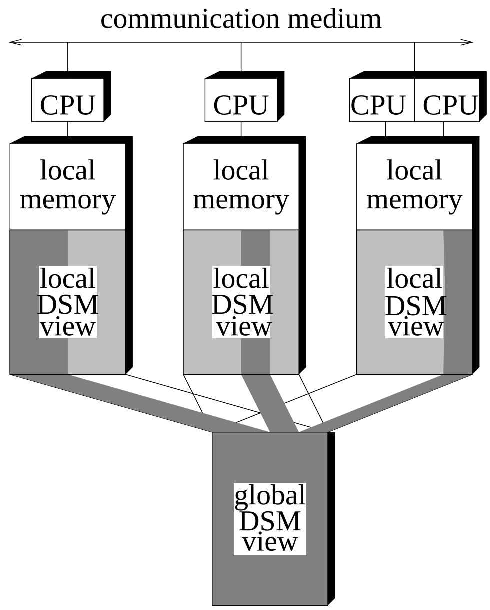

Fig. 1. Schematic View of a DSM System

reserved for DSM data. While only a subset of all DSM pages may be accessible at a node at a given time (as depicted by the darker shading), this local DSM view is transparent to the user. To the user, the DSM portion of the addressing space seems as a globally consistent, shared data area. For simplicity, it is assumed for now that read and write accesses are not distinguished and that sequential consistency [14] is preserved, *i.e.* memory accesses are globally ordered and a page is owned by one and only one node at a time. These assumptions will be lifted later.

The operational model can be described as follows. Initially, each DSM page is owned by a single designated node. A page table is initialized on each node with the initial owner for each page. If a node accesses data within a page that it owns (darker shading in Figure 1), the data is readily available and the access has the same characteristics as a local memory access. Conversely, if an access references a page that the node does not own (lighter shading in Figure 1), the access is deferred and a remote page request is issued to the owner of the page. When the page is received from the remote node, the access is finally granted.

This exchange of pages is depicted in Figure 2. The pages currently not owned by a node are protected, *i.e.* an access to the page causes a memory fault, for example on node 1. The memory fault is reported to a worker, who determines the (probable) owner of the page and sends a request to the remote owner (node 2). This request is received by the communication server of node 2, where the local page table is consulted. If the node does not own the page anymore, the request is forwarded to its probable owner (node 3). If the communication server on node 3 determines that it still owns the page, it delegates a worker to act upon the page request. The worker invalidates (protects) the page on node 3 and sends the page content directly to the requesting node 1. The communication server of the requesting node 1 then delegates a (possibly different) worker to act upon the sent page. The worker validates the page on node 1 and enables access to the page. Had there been more intermediate nodes, the original page request would have been forwarded along a chain of probable owners. But in the end, the request would be handled by the actual owner (node 3 in the example) as described before.

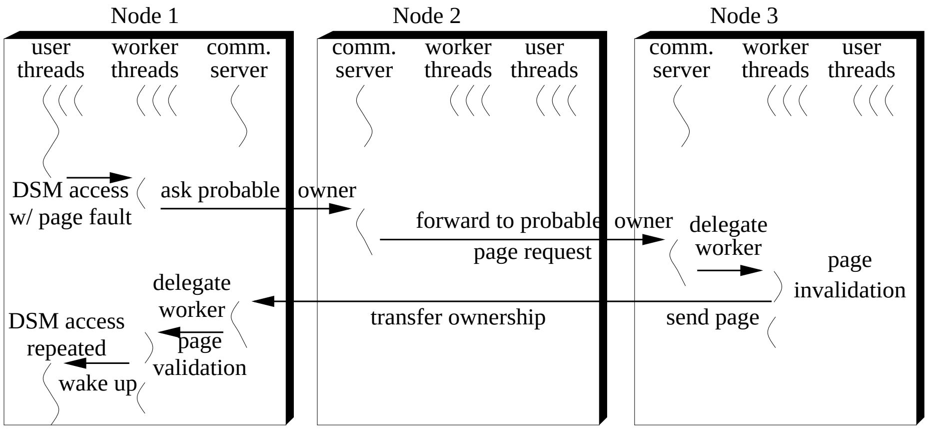

Fig. 2. Operational Model for Page Fault Handling

This model has been be implemented in an environment where nodes are represented by processes. The processes can be distributed arbitrarily on different workstations, possibly also with multiple processes on one workstations. Within each process, different threads of control may be created by an application on the user level while system-level threads include workers and the communication server. Worker threads are delegated to act upon page faults that occurred within a user thread. Among other requests, worker threads also handle page invalidations if the communication server receives a page request for a page that the current node owns. Generally, workers handle any requests that may involve more time-consuming actions. Similarly, the communication server is implemented as a separate thread of control. Its main purpose is to serve as a gate for incoming messages, handle less time-consuming messages right away and delegate more time-consuming requests to workers. This specialization of assigned tasks provides better responsiveness than a monolithic single server model since an I/O-bound or communication-intensive thread blocks until the I/O has been performed while other threads may continue to run.

3 Ada 95 Shared Passive Library Units

The distributed systems annex of the Ada 95 LRM [8] distinguishes between *active* partitions, which are equivalent to a regular Ada 95 program with one or more threads of control (environment task plus user-defined tasks), and *passive* partitions without a thread of control. A distributed program comprises a set of partitions, which can be mapped onto processing nodes (for active partitions) or storage nodes (for passive partitions).

A passive partition contains a set of *shared passive library units*, which are restricted on their references such that they can neither contain a remote access to data within active partitions nor require a thread of control. It furthermore defines shared passive library units as a means for "managing global data between active partitions".

A passive library unit is preelaborated and can be syntactically specified by the `pragma Shared_Passive[(library_unit_name)];` . Additional legality rules restrict such a library to essentially the declaration of constants, types, data, and protected types without entries.

4 Mapping Passive Partitions onto DSM

By mapping the Ada 95 model onto the DSM model, the data declared within a shared passive library unit can be represented as the DSM data residing on (possibly protected) pages. The initial owner of a page is the passive partition that the shared passive library belongs to. The DSM system threads (workers and communication server) become an integral part of the runtime system.

4.1 Multiple Reader Protocol

A DSM with a single owner consistency protocol guarantees that at most one active partition may access the data of a passive library at a time. We call this model the single-reader/single-writer (SRSW) protocol. Alternatively, a multi-reader/single-writer (MRSW) protocol may be used in conjunction with data declarations within passive partitions. MRSW provides either read access to the same DSM page (and hence the associated data) of more than one node at a time or write access for exactly one node. Multiple readers form a so-called *copy set* that is represented as a tree of nodes rooted in the first reader. Consider the left picture of example in Figure 3. As a node R requests read access, it may receive a copy of the page from any node in the copy set (say node C). Node R is subsequently inserted into the copy set by node C, *i.e.* node R becomes a child of node C (right picture of Figure 3).

When a node W requests write access, it contacts the root node T of the copy set (*via* a chain of probable owners), also depicted on the right side of Figure 3. Node T then sends a request to invalidate the page (represented as dashed lines) to all its immediate children. Each node C in the copy set propagates this request to its children, invalidates the page, awaits the children's acknowledgment (unless it is a leaf) and sends an acknowledgment to its parent (dotted lines). When T receives an acknowledgment, all copies of the page have been invalidated. Thus, T may now transfer ownership of the page to node W, the requesting writer. The MRSW protocol provides more potential for parallelism than the SRSW protocol at the expense of additional communication to invalidate a page. We may give the user the option to choose the protocol for data declaration with a default of the MRSW protocol (see [17] for more details).

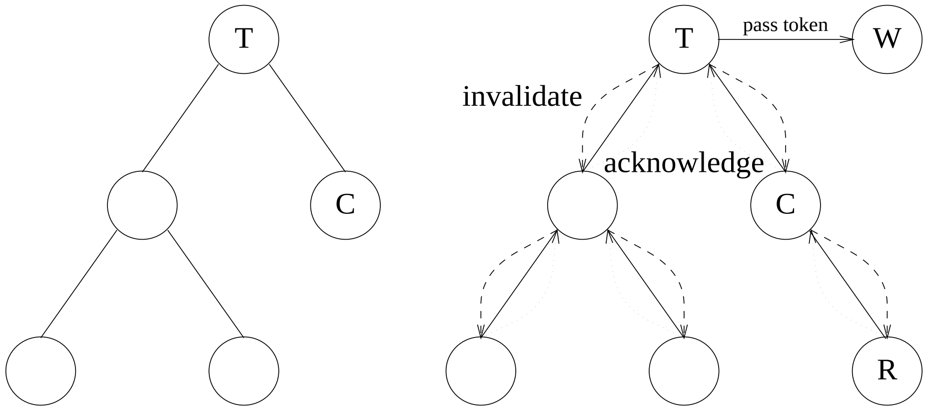

Fig. 3. Example for Multiple Reader Protocol

4.2 Protected Objects

The protocols discussed so far can be used in conjunction with regular data declarations within passive partitions. However, a sequence of accesses to some data (*e.g.* a read followed by an increment and a write) may not be consistent without synchronization since other active partitions may access the data in between. To guarantee mutually exclusive access, the data should be encapsulated in a protected object. Since protected types within shared libraries are limited to protected procedures and functions within the body, *i.e.* protected entries are not allowed, the protection can be realized by simple mutual exclusion as opposed to the more general "eggshell model" required for protected entries [7, 4]. However, a protocol for *distributed* mutual exclusion has to be provided since protected objects within the passive partition may be accessed from different nodes (active partitions).

A simple protocol for distributed mutual exclusion could be implemented on top of the SRSW protocol. A POSIX mutex object is declared as DSM data (see POSIX threads for details [23]). *Local* mutual exclusion *via* `pthread_mutex_lock` results in an attempt to lock this mutex. The access may trigger a page request through the SRSW protocol before the local lock may succeed. However, this simple protocol is implementation-dependent on the POSIX threads library since the internal structure of the mutex object is transparent and may contain references to the local Pthreads kernel. In general, this approach is non-portable, in particular for heterogeneous environments.

An improved protocol still utilizes the SRSW protocol but implements local mutual exclusion as an indivisible operation (*e.g., via* test-and-set) on a DSM data object. The access may also trigger a page request but the approach is portable, even for heterogeneous environments. Unfortunately, this protocol may not fully utilize potential parallelism since multiple readers (protected functions) could not access the protected object at the same time.

When the MRSW protocol is used in conjunction with an indivisible operation on DSM data, multiple concurrent readers could be permitted. But the MRSW protocol may admit new readers even after a write attempt has been issued for a protected object. In other words, accesses may not be granted in the same order that they occurred. This violates the Ada semantics of protected objects. Furthermore, accesses to a protected object may be issued at different priority levels. Clearly, a more sophisticated protocol is required.

We have designed a prioritized token-passing protocol for distributed mutual exclusion to address these issues [18]. The unprioritized version is a modified MRSW protocol where requests are served FIFO. Thus, multiple readers are only admitted for adjacent requests, *i.e.* if no write request was issued between the two read requests. The prioritized version tags probable owner chains with the priority level of former requests. This allows future requests to be forwarded along different chains depending on their priority. A request at the highest priority propagates straight to the actual owner (*via* some intermediate nodes). A low priority request travels to the last node L requesting access at the same priority or the next higher priority. The request is then stored in this node L. Thus, a distributed FIFO queue of requests is formed and mutual exclusion is granted for members of this queue in the prioritized order. Furthermore, the the protocol requires that additional information is stored along probable owners for "high" priority requests to avoid race conditions. However, the algorithmic details are beyond the scope of this paper (see [18]).

The data for a protected object is passed along with the token for mutual exclusion so that access faults and page transfers are not needed. Instead, the data is validated when the token arrives and invalidated when the token is passed on. A protected procedure requires an exclusive write lock within the token protocol. Simultaneous invocations of protected functions, on the other hand, can exploit their potential parallelism through concurrent read locks. Thus, the prioritized token protocol exhibits the required semantic behavior in terms of the Ada language combined with the benefits of parallel executions.

5 Integration into the Gnat Environment

The Gnu Ada Translator (Gnat) [24] supports the distributed systems annex of Ada 95 *via* the distributed system annex environment (DSAE), which is available separate from Gnat. DSAE consists of a tool Gnatdist [11] and a library Garlic [12]. Gnatdist is responsible for configuring the partitions of a distributed Ada 95 program. It supports the mapping of partitions onto nodes and drives the compilation and linking of the different components into one program per partition, *i.e.* it supports the post-compilation semantics of the distributed systems annex. Garlic is a library, which provides the interface between the network and the distributed application. A central synchronizer server handles partition identification and requires that each node register with this server upon elaboration. The library also supports remote subprogram calls that correspond to the `pragma Remote_Call_Interface[(library_unit_name)];` and synchronizes the termination of the distributed nodes. However, DSAE does not support shared passive library units.

DSAE and the mapping of passive partitions onto DSM can be combined in one system as depicted in Figure 4. The DSM system provides its own runtime library built on top of POSIX threads [23, 19]. The DSM system can be readily integrated since the Gnu Ada Runtime Library (GNARL) also relies on POSIX threads and does not require any modifications for DSM inclusion (except for DSM initialization). Once the DSM runtime system is initialized, it spawns a pool of workers and a communication server on each node to support a DSM system efficiently in a decentralized fashion. In contrast, DSAE still relies on a centralized synchronization server, which may create a bottleneck. Nonetheless, the Garlic library is used for partition handling, *i.e.* distributed creation and termination.

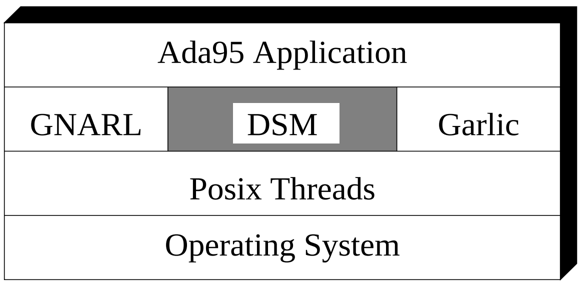

Fig. 4. Layering of Runtime-System Components

Currently, all passive library units reside on one node upon program start and are distributed upon request, *i.e.* whenever a page fault occurs on a remote node. However, the Gnatdist tool could be enhanced to support the distribution of passive library units onto arbitrary nodes. This would require the generation of a mapping table between host names and page addresses corresponding to the data of the passive library units. Minor modifications of the linking process should be added to include passive library units and the DSM library. Furthermore, the Gnat compiler currently ignores the `pragma Shared_Passive`[`(library_unit_name)`]`;` and needs to be enhanced to check for semantic restrictions on library dependences. The code generator needs to emit different code for protected objects within passive libraries. Instead of calling mutual exclusion routines of the Ada runtime system, mutual exclusion primitives of the DSM system need to be invoked to provide mutually exclusive access on a distributed level. Protected functions of a passive partition acquire a distributed read lock while protected procedures acquire a write lock according to the prioritized token-based protocol for distributed mutual exclusion described in the last section.

6 Future Work

Current work includes the implementation of the protocol for prioritized token-based distributed mutual exclusion. Furthermore, support for Ada 95 passive libraries on heterogeneous distributed environments is being added in a portable way. So far, heterogeneous support within our environment has been realized for C programs, only. Heterogeneous system support requires modifications to the Gnatdist tool to support data conversion between different architectural formats. A number of improvements are being worked on in the DSM runtime system to improve throughput by reducing the amount of required communication, including the support of different coherence protocols and the distinction between read and write accesses. These issues are fully transparent to the user and the Ada 95 system, including the DSAE subsystem.

7 Related Work

DSM systems have been an increasingly popular area of research. Several overviews compare different systems [21, 22]. Software implementations of DSM, such as discussed in our paper, can be distinguished by their reliance on modifications of the operating system, source language, compiler, and runtime system. We focus strictly on runtime system modifications, since we are most concerned with a portable implementation of an existing language construct (Ada 95 shared passive library units). These systems handle data sharing at the page level using page protection mechanisms as described in our work. TreadMarks is a page-based system supporting different consistency protocols and multiple writers [2]. Midway also supports multiple consistency protocols, which can be changed dynamically [5]. Filaments employs a set of lightweight worker threads (similar to our worker threads) to act upon remote processing requests quickly, using adaptive schemes to dynamically reconfigure data placement [15]. The consistency models of these system require explicit synchronization points within a program. Our approach uses a stronger consistency model without explicit synchronization points, which avoids modifications to the compiler and facilitates the integration into existing Ada environments. Other DSM systems handle data at the level of objects (*e.g.* Orca [3]), tuples (*e.g.* Linda [1]), or at variable sizes (*e.g.* Munin [6]).

Midway's entry consistency associates DSM data with locks and transfers the data when locks are passed between nodes [5]. This model is resembled by our handling of protected objects within passive partitions. However, we use a *prioritized* protocol for distributed mutual exclusion while their protocol resembles the much simpler SRSW protocol adapted for mutual exclusion. Mutual exclusion along the lines of the SRSW protocol was first described by Naimi *et. al.* [20]. The MRSW protocol resembles the decentralized dynamic manager algorithm by Li and Hudak [13]. Notice that Li and Hudak's algorithm reduced to writers only is equivalent to the token-based algorithm by Naimi *at. al.*.

A software component to support DSM for Ada was proposed by Kermarrec and Pautet [9]. Their implementation consists of primitive operations to manipulate the distributed shared data, such as peek, poke, copy, alloc, and free. Also, an outline for a generic shared memory manager is given, which can be instantiated for different data types to address the strong typing of the Ada language. An automata describes the transitions between DSM operations. However, the paper does not give enough detail to relate primitive operations to the handling of data pages. It is not quite clear to which extent the proposed system has been implemented or how it might be integrated with an existing Ada compiler.

The DSM variant of a tuple space, as implemented in Linda, has been discussed in the context of Ada. The design issues and interface specification have been discussed for implementing Linda over Ada 83 [10] and Ada 95 [16]. The latter work has also been extended to discuss how Ada 95 passive packages could be supported by such a Linda system [25]. A preliminary library has been developed but the authors admit that the layering over Linda is likely to induce a considerable performance overhead. They suggest an intermediate compilation step to transform passive library units into Linda-like Ada packages to support heterogeneous systems in the future. An integration in the Gnat environment would require changes at the compiler or pre-processor level. In contrast, our DSM system can be readily integrated into Gnat and can be extended to heterogeneous systems almost without compiler modifications. (The only modification would be to require a uniform data alignment for different architectures, which can be easily integrated into the back-end of the compiler.) Otherwise, simple modifications to the Gnatdist tool suffice.

8 Conclusion

We have shown how Ada 95 passive library units can be implemented *via* distributed shared virtual memory. A DSM system operating at the granularity of pages can be implemented in a portable fashion with an independent communication subsystem. Protected objects can be supported by our prioritized protocol for distributed mutual exclusion. A sample DSM system has been designed and implemented over POSIX threads. It has been integrated into the environment of the Gnu Ada Translator. Due to the independent design of the DSM system, including its own communication layer, the system integration requires minimal changes to the Gnat environment. To the user as well as the Ada runtime system, the DSM system remains entirely transparent. This is the first implementation of shared passive library units within an Ada 95 compilation environment, to our knowledge. The resulting system experiences a communication overhead for the first access to shared data not present on the current node since that data has to be requested from a remote partition. Once the ownership of the DSM data has been transfered to the current node, consecutive accesses take the same time as local memory accesses. Thus, for multiple, consecutive accesses the communication overhead only has to be paid once for data accesses within passive partitions implemented *via* DSM, where a message-passing or RPC implementation would have to pay the communication overhead for each data reference. Using the DSM paradigm, distributed applications can be designed to access distributed shared data efficiently by avoiding (or at least reducing) simultaneous / interleaved access to this data from different partitions. Furthermore, the shared memory model inherent to the Ada language can be used without modifications in a distributed environment.

References

1. S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8), May 1986.
2. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.
3. H. Bal and A. Tanenbaum. Distributed programming with shared data. In *International Conference on Computer Languages*, pages 82–91, 1988.
4. J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.
5. B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.
6. J. Carter, J. Bennet, and W. Zwaenepoel. Implementation and performance of munin. In *Symposium on Operating Systems Principles*, pages 152–164, 1991.
7. E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.
8. ISO/IEC. *ISO/IEC 8652L 1995 (E), Information Technology – Programming Languages – Ada*, February 1995.
9. Y. Kermarrec and L. Pautet. A distributed shared virtual memory for Ada 83 and Ada 9x applications. In *TRI-Ada*, pages 242–251, 1993.
10. Y. Kermarrec and L. Pautet. Ada-linda: A powerful paradigm for programming distributed Ada applications. In *TRI-Ada*, pages 438–445, 1994.
11. Y. Kermarrec and L. Pautet. Gnatdist: A configuration language for distributed Ada 95. In *TRI-Ada*, 1996.
12. Y. Kermarrec, L. Pautet, and S. Tardieu. Garlic: Generic Ada reusable library for interpartition communication. In *TRI-Ada*, pages 263–269, 1995.
13. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions of Computer Systems*, 7(4):321–359, November 1989.
14. J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.
15. D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, August 1996.
16. K. Lundqvist and G. Wall. Using object oriented methods in Ada 95 to implement linda. In *Ada Europe*, 1996.
17. F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Parallel and Distributed Runtime Systems*, April 1997.
18. F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *Workshop on Parallel and Distributed Real-Time Systems*, April 1997.
19. Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.
20. M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing*, 34(1):1–13, April 1996.
21. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.
22. J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.
23. Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].
24. New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.
25. G. Wall and K. Lundqvist. Shared packages through linda. In *Ada Europe*, 1996.