Published in *Proc. of the 9th International Parallel Processing Symposium (IPPS'95)*, Santa Barbara, California, April 1995.

Run-Time Support for Asynchronous Parallel Computations

Guillermo A. Alvarez *   Marcelo O. Fernández †   Rogelio A. Alvez †
Sylvia Rodríguez †   Julio A. Sánchez Avalos †   Jorge L. C. Sanz †

* Dept.of Computer Science and Engin.,Univ.of California-San Diego,La Jolla,CA 92093-0114 – galvarez@cs.ucsd.edu

† IBM Argentina,Ing.Butty 275,1300 Buenos Aires,Argentina – {roge,sanz}@buevm1.vnet.ibm.com

Abstract

In this paper we describe DREAM, a distributed environment that provides run-time support for parallel computations on asynchronous multiprocessors. The system supports global distributed arrays as collections of subarrays in the local memories of the intervening processors. Nodes allocate and deallocate array portions dynamically, and access external array sections without the intervention of the user program running in the remote node. Remote accesses can be performed while the program continues its execution, thus overlapping communication and computation. This feature allows the user to implement dynamic communication patterns by accessing external array elements on demand without incurring a heavy performance penalty. DREAM also vectorizes requests into larger network messages for efficiency. We report performance results for an application running on top of a prototype of the system, showing good scalability and masking the network latency with computation.

1 Introduction

This work describes the motivations and major design aspects of the Distributed Run-time Environment for Asynchronous Multiprocessors [1]. DREAM is a distributed implementation of shared-memory arrays integrated with pairwise and collective message passing facilities. In order to mitigate the overhead of communication startup and latency, it implements message vectorization in a user-transparent way, and it promotes a communication style in which latency is overlapped in time with useful computation. Accesses to nonlocal array elements are requested and executed on demand, thus implementing dynamic communication patterns (usual in irregular problems) efficiently.

DREAM has been designed as a set of mechanisms. It is up to the client application to implement particular policies for higher-level concerns (e.g. naming, data distribution and ownership), for their effectiveness varies according to the problem at hand. Our approach is to provide intermediate-level services that are usable by different programming models, and portable among different underlying architectures. Shared memory, message passing and data parallelism are examples of traditional programming paradigms that have been treated independently in many cases in the past. DREAM furnishes a set of tools that support these paradigms without enforcing any of them through design constraints. It is a run-time support facility for compilers of parallel programming languages, which can generate sequential object code interspersed with library calls to DREAM for each node of an asynchronous multiprocessor. On the other hand, the primitives can also be invoked from hand-coded applications.

The outline of this paper is as follows. Section 2 describes DREAM's architecture and design. The existing prototype and the current version of the C interface are described in Section 3. Section 4 contains performance measures for an application coded on top of the prototype. A summary of the related work is provided in Section 5. Finally, in Section 6 we draw some conclusions.

*Guillermo Alvarez was partially supported by a doctoral fellowship from the Organization of American States.

2 Overview of DREAM

DREAM is targeted to sets of interconnected processors executing independent instruction streams. It can be implemented on a wide range of architectures, from clusters of loosely coupled workstations (where a prototype has already been developed) to MIMD multiprocessors.

A parallel application running on DREAM consists of a set of processes called *Node Programs* (NPs, one per participating processor), that communicate by sending messages and sharing distributed data structures. Each NP runs asynchronously unless synchronization is explicitly requested by invoking the barrier primitive. The NPs can access nonlocal array elements on demand through the *Distributed Memory Support* (DMS) facilities. This style of communication relies on two server processes: *Message Generation Server* (MGS) and *Message Attention Server* (MAS). If desired, each NP can also keep private variables, which will not be accessed from remote nodes. In addition, we provide message-passing (*Node-to-Node*) facilities, that are intended to be used for direct exchange of anonymous data between NPs. Figure 1 shows a diagram of DREAM in a generic node.

2.1 Distributed Memory Support

Nonlocal array elements and sections are accessed by specifying both the processor where they reside,

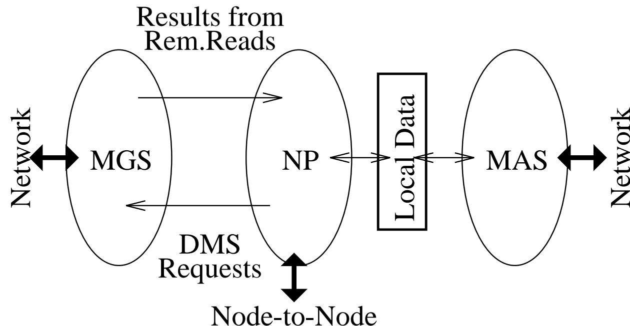

Figure 1: Structure of DREAM in a node

and their local names in that processor. In general, whenever an NP determines the need of a remote access, it calls the corresponding DMS primitive; no prior, compile-time knowledge of the communication pattern is assumed. Therefore, the DMS is the right choice for dynamic communication patterns, and for the exchange of named array elements between nodes.

Three types of operations are provided: read, write, and remote write. By invoking a remote write, a node instructs another node to write a data value on a third node; the issuing node provides the name of an array element owned by the second node, and the place of the third node where the corresponding (unknown) value must be written.

Orthogonally, the DMS primitives come in both blocking and nonblocking versions. After invoking a nonblocking operation, control returns to the NP, who can keep executing while DREAM performs the remote access. Communication overhead is thus masked with computation. For pending reads, the NP polls the DMS later to test the completion of the access, and then accesses the actual data. The blocking (*Rush*) primitives do not return the control to the calling NP until the operation has finished.

As an example, let `A`, `B`, and `C` be distributed unidimensional arrays; furthermore, assume that `A` and `C` have the same shape and distribution. The array expression `A = B[C]` can be executed by issuing read requests from the owners of each `C[i]` to the nodes that hold `B[C[i]]` (in order to get the elements of `B[C]`), and then performing the local assignments onto `A[i]`. Although "owner computes" was used in this case, arbitrary scheduling policies are supported by our approach.

Additional operations (like remote increments) can be defined. They would be performed with a single network access, instead of separate read and writes with explicit lock management to guarantee atomicity.

The MGS receives DMS requests from the user application, and sends the corresponding messages to the remote nodes in order to have them serviced. If a request generates a reply from the peer side (i.e. for reads), the MGS delivers the results to the NP on demand. The NP executes blocking or nonblocking polls to learn about the completion of DMS operations.

Our current implementation maintains a system of queues at each processor, shared between the NP and the MGS. It contains one queue for each remote NP participating in the run. When the NP invokes an asynchronous DMS primitive, a request is inserted in the outgoing queue of the corresponding node. The MGS packs the contents of each queue into a single message to amortize the network startup cost over a longer packet. The contents of a queue are sent out when the queue size reaches a certain threshold that can be set by the user depending on the communication medium and on the particular application. To avoid starvation, the MGS sends the contents of the queue after a certain amount of time, regardless of its size. Moreover, the user can explicitly flush the queues by invoking a DMS primitive.

Another way of improving efficiency is to overlap communication with computation by using nonblocking DMS facilities. The MGS can exploit any facilities supported by the operating system at each particular platform. In certain architectures, the MGS would issue nonblocking message sends, and post the buffer addresses for nonblocking receives. Specialized communication hardware, like dedicated communication processors and DMA controllers, would do the buffer transfers and run the protocols while the main processor proceeds with the NP.

Two identifiers are associated with each asynchronous read request: *message type* and *request id*. Message type is specified by the NP, and is used for mailbox searches on the replies. It provides a user-controlled way of grouping related requests from the application's viewpoint. The request id is chosen by DREAM to identify each particular request (regardless of its message type), and is reported to the NP when a read operation is invoked.

Rush primitives originate a network message as soon as they are invoked. No queueing system is necessary due to their blocking nature.

The MAS services external requests coming from remote MGSs. When a read must be done, the MAS fetches the data and sends them to the MGS of the requesting processor (for a read request) or to the MAS of a third processor (for a remote write request). For write requests, the MAS performs the corresponding update on the elements acting directly on the processor's memory. No network reply is necessary for writes, except for Rush services where the remote NP must be unblocked. In this way, requests from other nodes don't perturb the NP that owns the accessed data. The servicing policy ensures fairness, and a higher priority for incoming Rush requests. Array elements are accessed by the local NP and MAS in parallel. Operations are guaranteed to execute atomically[1] at the array element level, to rule out abnormal behaviors resulting from concurrent interactions.

Distributed arrays are allocated and deallocated dinamically by the coordinated action of every processor which holds portions of them, not by invoking network-wide primitives. Using the functions that provide information on the current configuration, the

[1]Parallel operations on the same array element behave as if done sequentially in an unspecified ordering.

NPs can determine the initial data allocation (e.g. based on the set of available machines), and later balance the load dynamically by reallocating the data on the fly. DREAM furnishes a tuple database (described in Section 3) that can be used by the NP to keep track of the current state of the system, as is done in [2] with a run-time symbol table. Using these features, a smart NP can run without even being recompiled on a variable-sized network where the loads of each machine are unpredictable. However, DREAM doesn't address the data ownership, load balancing, and process migration issues mentioned above; it only provides the tools for implementing them.

2.2 Node-to-Node Communication

These primitives exchange anonymous data in a loosely synchronous way, either between two processors or among all the intervening nodes. Only values are transferred directly between NPs, with no notion of names or array elements. DREAM provides blocking and nonblocking sends and receives, and also blocking and nonblocking broadcasts. Correct delivery, sequencing, and lack of duplicate messages are guaranteed. Incoming messages are placed in an input queue, and distinguished by the message type field as criterion for mailbox searches.

The Node-to-Node set of primitives can be used when the communication pattern is fixed at code-generation time. When the set of external elements accessed by each node is known, the owner of the data can send them to every node that needs them, without being asked. This strategy cuts in half (modulo message grouping) the communication steps when compared to DMS read requests.

3 Prototype

We have developed an implementation of DREAM on a network of IBM RISC System/6000 workstations running the AIX operating system. Application programs are written in the C programming language, and invoke DREAM services via calls to library functions. Communication is accomplished by using sockets (TCP for asynchronous requests, UDP for Rush requests). The servers (MGS and MAS) are implemented as processes that run concurrently with the NP on each node.

A description of the most important primitives of the current C interface follows. To initiate a running session, every processor invokes `InitDream`; similarly, `CloseDream` ends the session and frees the locally allocated resources. `DreamCtl` primitive allows the user to fine-tune internal parameters of DREAM (e.g. the water mark that causes a queue to be flushed), thus tailoring its behavior to each particular application.

A node creates a globally accessible local array by invoking `AllocArray` with the type of its basic elements, the number of dimensions, and the extent along each dimension. The allocation primitive returns an array identifier (optionally fixed by the user) to be used in subsequent operations. Symmetrically, `DeallocArray` destroys a local array. Elements of local arrays are accessed via a set of primitives that closely resemble the operations of the DMS explained below. The `SwapArrays` primitive exchanges the contents of two local arrays of the same shape and type in constant time; this is useful for implementing the update step of iterative algorithms.

We single out `RReadInt(Node,MsgType,ArrayId, Coord1,...,Coordn)` as a representative of the DMS asynchronous primitives. It initiates a nonblocking remote read of a single element (an integer in this case, though the current interface supports shorts, floats and doubles as well) of an n-dimensional array located in another node. The set of coordinates must be the local name of the array element in the remote node. The calling NP continues executing; the primitive returns a request id for the pending operation, that added to the user-defined message type will be used to retrieve the answer after the operation finishes. The `RWrite`*Type* call is similar, but it includes the value to be written, and it doesn't return a request id because no answer is needed by the calling NP in that case. Rush versions don't involve request ids because they are blocking. All the DMS primitives are also available in set versions, which access a whole section of a remote array with a single DREAM operation. Sections are denoted by specifying a triplet start:end:stride for each dimension of the array.

DREAM provides a set of primitives to test the completion of remote read operations, and to access the data returned by them. For instance, `RespAvail` tests if any response from a given message type is available without blocking the NP, and `WaitForResp` blocks the caller until any answer of the given message type arrives. `RespPending` tests if there exists any read operation in progress. When an answer is actually present, `RespArrived` gives its request id and size to the caller. The NP can then either copy the data to its own address space, or get a pointer to it. We also provide a database that, given a numeric key, stores (`Remember`) and retrieves (`Remind`) a tuple of numbers. An evident use for this database is to store the relationship between the request id of a pending read operation, and the array coordinates of the corresponding elements that will be received; however, the NP can use it for other purposes as well. `Synch()` blocks the calling NP until the remaining nodes have also called it. Every DMS request initiated before calling `Synch()` is guaranteed to be serviced before any request posterior to the call; thus the user can assert properties of the current state even in the presence of pending operations. The C interface contains more primitives for session and queue control, Node-to-Node communication, and information on the state of the run that are not explained here for brevity.

4 Performance: Regular Grid

To evaluate the performance of the prototype, we implemented a Jacobi iterative solver on a two-dimensional rectangular grid with a nine-point stencil. The grid is partitioned in rectangular blocks among

the nodes. To update its border elements, each processor keeps local replicas of the border elements of its neighbors. These border updates must be deferred until the replicas have the current values. A global barrier has two disadvantages in this case: it has a high message cost, and it involves all the nodes, effectively making every node wait for the slowest even if it isn't its neighbor. A pairwise (or more general) group synchronization can be easily implemented in DREAM, by using flags and nonblocking remote write operations. In this example, each node sends the border elements to its neighbor, and then notifies this event by setting a flag in the neighbor NP's address space. Since operations are nonblocking and the order between the writes is preserved, this scheme effectively achieves a synchronization between the two nodes using a single network message in most cases.

The algorithm was tested on a 16 Mbit/sec Token-Ring network of IBM RISC System/6000 workstations running AIX 3.2. To mask the communication latency with computation on local elements, the border elements are sent out first, then the central elements are updated, and then the received replicas are used to update the border elements. Because the workstations used have different processor speeds, the data distribution is intentionally uneven, so that the load is well balanced and idle times are minimized.

Figure 2 shows the results obtained for each number of processors and problem size. Times are expressed in seconds. The table contains the speedup (average running time on each workstation alone, divided by running time of parallel execution), and the inter-wave delay (total idle time along the run spent waiting for replicas to arrive). The problem size is made larger for larger N to compensate for the effects of virtual memory pagination, avoiding a too small portion of the grid at each intervening node. Inter-wave delays are completely negligible for N=2,3,4 (if not zero at our level of precision). Due to the restricted class of partitions supported by our implementation of the Jacobi algorithm, the quality of the load balancing degrades for N=5,6. The different processor speeds become more of an issue in these cases. Therefore, idle times are larger for the last two entries. The measurements support our claim that idle times can be made low due to the overlapping between communication and computation achieved by DREAM. Also, the use of a weak (pairwise) form of synchronization relaxes global constraints and contributes to decrease the delays.

The graph shows the speedup as a function of the number of nodes. The dashed line represents the ideal speedup; the performance of this application is close to this value, though somewhat smaller as usual for larger values of N. The numbers obtained exhibit good scalability; the grid partitions were determined by trial and error, and are surely improvable with a more exhaustive approach.

5 Related Work

There are basically three categories of relevant previous works in the literature. Systems like PVM [3] rely on basic, collective message-passing for communication. In general, the provided facilities are lower-level than DREAM's. Grouping of messages into larger packets is handled explicitly by the programmer, and there is no notion of distributed arrays — only anonymous sequences of bytes are exchanged.

Compiler proposals like Fortran D [7] and Data Parallel Fortran [6] extend sequential programming languages to make parallelism explicit in the source code. Split-C [5] and XDP [2] address a set of optimizations similar to DREAM's at the source and intermediate code levels, respectively. These systems and compilers can rely on DREAM as a lower layer for run-time support. Others rely on a careful balance between the expressive power of the syntactic constructs, and the effective limits of the underlying compilation techniques: Kali [8] and PARTI [4] promote a compute-communicate programming strategy when the communication patterns are unknown at compile-time. Global communication is used to determine the read/write sets of each processor; only after that can computation begin, as opposed to the on-demand approach followed by DREAM. The work developed by Chen [9] also allows on-demand nonlocal accesses; however, the mechanism is blocking for the requesting node, and the remote side is interrupted to service the requests.

Distributed shared memory systems like IVY [10] provide memory sharing at the page level with read-only replication. The user can not influence the distribution of pages, nor tailor the data consistency control to the application.

6 Conclusions

Targeted primarily to distributed memory architectures, DREAM provides both an intermediate-level distributed shared memory, and message-passing facilities. Special emphasis has been placed on making interprocessor communication explicit at the interface level, and on the absence of fixed policies implied by design decisions. The goal of the system is to supply a set of mechanisms that the client application can use to implement a variety of possible policies without incurring performance penalties.

Computation is decoupled from ongoing communication at the interface level, thereby allowing the user to write code that overlaps both activities to mask the network overhead. Requests are vectorized into a single network packet whenever possible to amortize the message startup delay. Irregular problems usually give rise to dynamic communication patterns; DREAM solves them by accessing nonlocal data on demand whenever the need is detected. Only the source and destination nodes are involved in the processing of each request. The numbers obtained with the sample numeric application show that the prototype scales well, and that the decoupling can indeed lead to negligible idle times.

Nodes	Problem Size	Int-wv.Delay Max.	Int-wv.Delay Avg.	Sequential Time	Parallel Time	Speedup
2	1000 x 1000	0.000	0.000	37.728	19.025	1.983
3	1000 x 1000	0.112	0.037	42.024	14.599	2.878
4	1000 x 1000	0.000	0.000	40.022	10.967	3.649
5	1200 x 1200	1.904	0.380	57.910	14.109	4.104
6	1200 x 1200	1.444	0.240	56.500	11.340	4.982

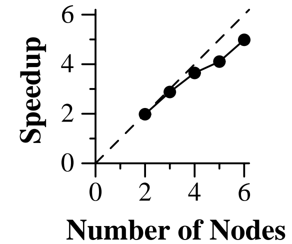

Figure 2: Performance Measurements for the Regular Grid Application

Acknowledgements

We would like to thank Allan Snavely and the anonymous reviewers, who provided useful suggestions for improving this paper.

References

[1] G. Alvarez, R. Alvez, M. Fernandez, J. Sanchez, and J. Sanz. Distributed Run-Time Support for Data Parallel Programming. Technical report, IBM Argentina, June 1992.

[2] V. Balasundaram, J. Ferrante, and L. Carter. Explicit Data Placement (XDP): A Methodology for Explicit Compile-Time Representation and Optimization of Data Movement. In *Proc. of 4th ACM PPoPP*, May 1993.

[3] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A Users' Guide to PVM Parallel Virtual Machine. CS-91-136, Univ. of Tennessee, July 1991.

[4] H. Berryman, J. Saltz, and J. Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines. *Concurrency: Practice and Experience*, 3(3), June 1991.

[5] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing '93*, November 1993.

[6] P. Elustondo, L. Vazquez, O. Nestares, G. Alvarez, J. Sanchez Avalos, C.-T. Ho, and J. Sanz. Data Parallel Fortran. In *Proc. of the 4th Frontiers of Mass.Par.Comput.*, October 1992.

[7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D Language Specification. TR:90-141, Rice University, December 1991.

[8] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[9] J. Li and M. Chen. Generating Explicit Communication from Shared-Memory Program References. In *Proc. of Supercomputing '90*, November 1990.

[10] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.