Distributed Run-Time Environment for Data Parallel Programming

Rogelio A. Alvez [*] Marcelo O. Fernández [†] Julio A. Sánchez Avalos [*]
Guillermo A. Alvarez [†] Jorge L. C. Sanz [*‡]

December 7, 1995

Abstract

In this paper, the fundamentals of a distributed run-time support for data parallel languages are described. This system, dubbed DREAM (Distributed Runtime Environment for Advanced Multicom puters), provides node-to-node communication, relaxed synchronization, data distribution independence, local naming, distributed memory support, and portability. These features are intended to solve both compile-time and run-time communication patterns efficiently.
A first prototype has been implemented on a network of IBM Risc System/6000 workstations. DREAM is a joint project between IBM Argentina and IBM Almaden Research Center, and will provide a run-time environment for a distributed implementation of Data Parallel Fortran [4].

Extended Summary

The success of data parallel programming does not depend only on compiler technology but also on an adequate run-time support. Node-to-node communication, synchronization, distributed memory support, and portability are key subjects in the design of such a run-time support. These features are the base on top of which a data parallel language should lie, and are the goals pursued in DREAM (Distributed Runtime Environment for Advanced Multicomputers).

Compiler technology is concerned with the problems of data distribution, global to local reference translation and synchronization points. Implicit information about the source program's behaviour can be gathered through data dependency analysis, and explicit information from parallel primitives [5]. This knowledge is used to improve efficiency by enf orcing locality of data references and to schedule computations that can be carried out in parallel over different processors.

A major trend in current research consists of designing clever extensions of serial languages for parallel machines. Some examples of this approach are Data Parallel Fortran [4], Fortran 8X [1], C* [9], Fortran 90 [8], Data Parallel C [6] and Fortran D [5]. Data parallel programs are usually written in a host language (usually Fortran- or C-like) plus parallel primitives for data distribution and scheduling. DREAM has been designed as a potential resource for the implementation of distributed Data Parallel Fortran [4] and other data parallel languages, and it allows usual features of these languages to be translated to a sequential program with library calls to DREAM. The resulting program, linked with proper libraries is replicated in every processor as a set of node programs.

DREAM is formed by a set of processes scattered onto the multiprocessor architecture. Three processes compose each node's structure. One of them, the Node Program, is the program generated by the compiler or

[*]Computer Research and Advanced Applications Group, IBM Argentina, Buenos Aires, Argentina.
[†]ESLAI, Escuela Superior Latinoamericana de Informática, Buenos Aires, Argentina.
[‡]Computer Science Dept., IBM Almaden Research Center, San José, California.

end-user. The NP uses the services provided by DREAM for node-to-node communication and local data access. The other two processes support the interaction with the other processors for distributed memory support. The Message Generation Server (MGS) has two main responsibilities: it sends messages with requests for external data, and passes the answers back to the NP. The Message Attention Server (MAS) receives messages with requests, performs desired functions (READ, WRITE, etc.) on local data and returns answers if necessary. Consequently, the NP is freed from the responsibility of sending local data to other requesting NPs. The NP that needs an external value sends a request to the processor where the data is allocated (through the MGS) and the MAS in the destination processor services the request.

The node-to-node communication facilities provided by DREAM are used to implement operations over distributed data when the compiler knows in advance which non-local elements are going to be needed in each node. These primitives enable processors to send and receive data to and from peer processors both synchronously and asynchronously. Also, some more complex aggregated primitives (such as broadcast and multicast) are provided. Node-to-node services must be designed by keeping the implementation independent from underlying communication protocols, making it portable over different systems.

On the other hand when a compiler cannot discover the communication pattern, a run-time analysis has to be made in order to use the node-to-node communication facilities. In the approach followed by Kali [7], PARTI [10, 11, 12] and Fortran D [5], collective communication takes place only before the execution of the parallel sections of a program. This is accomplished through the execution of an inspector code, which discovers the external data references in the subsequent parallel section(s) and communicates these requests to their repositories. This compute-communicate approach does not take advantage of the communication medium, since it is intensively used in some phases and idle in others. The inspector also implies a great global communication effort (to identify the data that each processor must send to others), and is not suitable for multiple levels of reference indirection. The communication style is imposed by the inspector-executor approach and does not depend on the nature of the application.

When off-processor references are resolved only at run-time, each parallel process has to determine the set of non-local elements required for the corresponding computation and get them from their repositories. This need also arises when a dynamic data distribution is u sed in the programming language. The approach supported by DREAM is based on having each processor ask non-local elements by demand, and is achieved through some distributed memory support facilities. These services do not incur in the overhead produced by an inspector and yield a more balanced use of the commu nication network. This asynchronous communication model is not intended to transfer data between user programs, but to access external elements of distributed structures allocated in other nodes. A fully asynchronous semantics at the language level, in which race conditions w ere allowed, could be supported by using DREAM.

In order to take advantage from the available bandwidth, a request packing techn ique is used. Instead of considering a network transaction for each nonlocal element required, these requests can be packed into only one transaction. Since there is a fixed cost for the startup of a network access, this packing technique lowers the latency overhead. Also, communication is overlapped with computation. This technique leads to an asynchronous behaviour (i.e. communicating processes that uncouple communication tasks from computation core). Processes that need to access external data do not have to block waiting for replies. They delegate this task on DREAM and continue processing.

A similar approach is proposed in [2, 3]. In [2, 3], howe ver, the mechanism for asking for external elements is blocking in the sense that computation will be suspended until necessary data is available. Furthermore, the peer side is forced to interrupt its execution in order to service the remote request. On the other hand, there is no packing strategy, thus implying a big cost due to the overhead produced by the startup of each external request.

The distributed memory support of DREAM gives a way to access the memory of a r emote host without communicating with the user program that runs there. This service is implemented in each processor by two processes (MGS and MAS) that handle the external requirements for local data. Packing strategies, network

control and protocol details are more suitable to be integrated in a separated module, in order to alleviate the user from these details. DREAM offers to the compiler a portable communication layer in the sense that the latter does not have to deal with packets nor with the task of packing messages to achieve communication performance.

Asynchronous data requests are directed to the node which holds the data elements that are to be accessed. DREAM accesses remote array elements using local naming, i.e. the selected elements are named in terms of the location they have within the owner's local memory. In other words, the compiler should solve the translation from global references to local references. These locations must be provided to DREAM, according to the data distribution and the communication pattern (which could also be static or dynamic).

Aside from asynchronous requests, DREAM provides facilities for urgent (RUSH) remote operations. A RUSH request share the same expressive power of the asynchronous requests, but it blocks the calling program until it completes. Because of their blocking nature, RUSH requests must be serviced by the DREAM components with the highest priority.

Remote operations can either perform a single remote access or a set of identical accessses on a regular subset of a distributed array. Three types of operations can be performed: *READ, WRITE* and *REMOTE_WRITE*. These operations are nonblocking, since the user code can continue computing on local data while the remote access is being performed. Also, some variations of WRITE and REMOTE_WRITE operations are provided, e.g. *ADD, DEC* and *MULT*. These operations were included because they are frequently used in arithmetic processing. A remote increment of a variable would otherwise be implemented through an external READ request, a local addition and an external WRITE request. Instead, a more complex ADD service specifies the increment in the same message and this operation can be remotely executed, thus avoiding one message and additional overhead. Some examples that illustrate the power of these operations are included below.

Example 1 Let arrays `A` and `C` be mapped in the same way [4]. The following array expression

$$\texttt{A} = \texttt{B[C]}$$

is solved by issuing READ requests to the nodes that hold the elements of `B` indexed by array `C` in order to get the elements of `B[C]` and then perform the local assignments onto array `A`.

Example 2 With the same assumptions the array expression

$$\texttt{B[C]} = \texttt{A}$$

could be executed analogously through the use of WRITE requests.

Example 3 Let arrays `B` and `D` be aligned. Then, the array expression

$$\texttt{A[B]} = \texttt{C[D]}$$

is efficiently solved with REMOTE_WRITE requests. With this operation the processors holding elements of `B` and `D` instruct the nodes where `C[D]` is located to send these data to the processors holding `A[B]`. The use of this facility substantially decreases the amount of network communication, using only two messages per data assignment.

During the execution of a given operation in parallel over many elements of distributed arrays, an efficient strategy for the NP is to ask for asynchronous remote services on nonlocal data and then compute upon local

data (while the nonlocal requests are being completed). In some cases the execution can not proceed until a given remote request has completed. The RUSH services can then provide an efficient (prioritized) completion in order to unblock the NP and resume execution.

Every node executes asynchronously until synchronization with other NPs is requested. DREAM includes a synchronization primitive that has a BARRIER effect between NPs, and also guarantees that remote and local operations initiated before the global commitment are completed before new operations are invoked. The implementation will take advantage of hardware synchronization facilities whenever they exist. This run-time service would be used by the compiler either when data dependencies between two different blocks of code are detected, or for debugging purposes.

An implementation of a prototype of DREAM on top of a network of IBM Risc System/6000 workstations running AIX has been developed. The current version of the prototype offers a C interface, allowing compilers for data parallel languages as well as end-users to invoke the DREAM primitives. The communication platform on top of which the facilities of DREAM lie is TCP/IP. The goal of this choice is to provide portability between different architectures. Also, some compilation techniques using DREAM services are being developed for a set of parallel constructs present in Data Parallel Fortran [4].

References

[1] E. Albert, K. Knobe, J. Lukas, G. Steele. "Compiling Fortran 8x Array Features for the Connection Machine Computer System". *SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, September 1988.

[2] J. Li, M. Chen. "Compiling Communication-Efficient Programs for Massively Parallel Machines". *IEEE Trans. Parallel and Distrib. Sys.*, pp. 361-376, Vol. 2, Number 3, July 1991.

[3] J. Li, M. Chen. "Generating Explicit Communication from Share d-Memory Program References". *Proceedings of the Fourth Internation al Conference on Supercomputing.* November, 1990.

[4] P. Elustondo, L. Vazquez, O. Nestares, G. Alvarez, J. Sanchez Av alos, C.-T. Ho, J. Sanz. "Data Parallel Fortran". Technical Report, IBM Almad en Research Center, Computer Science Department. March, 1991.

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

[6] P. Hatcher, M. Quinn. "Introduction to Data Parallel C", *Data-Parallel Programming.* The MIT Press, November 1991.

[7] C. Koelbel, P. Mehrotra. "Compiling Global Name-Space Parallel Loops for Distributed Execution". *Trans. Parallel and Distrib. Syst.*, Vol. 2, Number 4, October 1991.

[8] M. Metcalf, J. Reid. *Fortran 90 Explained.* Oxford University Press, Walton Street, Oxford, 1990.

[9] Thinking Machine Corporation. *C* Reference Manual, version 4.0A.*

[10] R. Das, R. Ponnusamy, J. Saltz, D. Mavriplis. "Distributed Memory Compiler Methods for Irregular Problems". ICASE Report No. 91-73, September 1991.

[11] H. Berryman, J. Saltz, J. Scroggs. "Execution time support for adaptive scientific algorithms on distributed memory machines". Concurrency: Practice and Experience, Vol. 3(3), 159-178. John Wiley & Sons, Ltd. June 1991.

[12] J. Saltz, H. Berryman, J. Wu. "Multiprocessors and run-time compilation". Concurrency: Practice and Experience, Vol. 3(6), 573-592. John Wiley & Sons, Ltd. December 1991.