Published in *Proceedings of the Sixth Swedish Workshop on Computer System Architecture (DSA'95), Stockholm, Sweden, June 1995.* pp. 65-66.

Virtual Shared Memory for PVM

Vladimir Vlassov, Hallo Ahmed and Lars-Erik Thorelli

Department of Teleinformatics
Royal Institute of Technology (KTH)
Stockholm, Sweden
vlad@it.kth.se

ABSTRACT

This article presents mEDA-2, an extension to PVM which provides Virtual Shared Memory, VSM, for inter-task communication and synchronization. mEDA-2 consists of functions to access VSM and a daemon to manage parallel program termination. Access to VSM is based on the semantics of the EDA parallel programming model. The aim of developing mEDA-2 was to facilitate construction of parallel programs in PVM by providing a unified approach to message passing and shared memory models.

1 Introduction

Most of the parallel programming environments, such as: PVM, P4, PARMACS, EXPRESS, MPI, etc., support the distributed memory programming model and are based on the message passing communication technique [1]. An alternative is the shared memory model which is characterised by its flexibility and ease of programming.

In this article we present mEDA-2, an extension library for the Parallel Virtual Machine, PVM [2], which provides Virtual Shared Memory, VSM. This is a dynamic set of shared variables used for inter-task communication and synchronization. Access to VSM is based on the semantics of the Extended Dataflow Actor model, EDA, which was developed at the Royal Institute of Technology, KTH, during the last few years [3, 4, 5, 6, 7, 8].

2 The EDA Multiprocessing Model

EDA provides a unified approach to communication and synchronization using distributed shared memory. A formal description of EDA and its first programming implementation can be found in [7, 8].

EDA objects are the executing units of an EDA program, which are realized in mEDA-2 as PVM tasks. Shared variables are distributed between objects where they can be accessed using synchronization rules defined in terms of *fetch* and *store* operations. Each shared variable may be in one of two states: *full* (containing data) or *empty*. EDA recognizes four kinds of shared memory operations:

(1) *x*-operations, for accessing critical regions in mutual exclusion and supporting synchronous producer-consumer relationships.
(2) *s*-operations, for supporting asynchronous producer-consumer relationships.
(3) *i*-operations, for synchronizing single writer-multiple readers and OR-parallelism.
(4) *u*-operations, for supporting asynchronous access to shared memory.

X-fetch and *x*-store operations are synchronous and alternating, they may also cause the executing object to be suspended. An *x*-store operation can store data only to an *empty* shared variable, otherwise, the store request is enqueued until the variable is emptied by extracting its value using a fetch operation. An *x*-fetch operation destructively reads (extracts) the value from the full shared variable. If the variable is empty the *x*-fetch enqueues the request on that variable until it becomes full by a store operation.

S-fetch and *s*-store operations facilitate stream communication between objects. An *s*-store operation is supported by a buffering mechanism. If the variable is already full, a new stored value is buffered until the variable is emptied. An *s*-fetch operation on a full variable extracts its value to local object memory.

An *i*-store to a full variable is ignored, while an *i*-fetch operation from an empty variable enqueues the request to that variable. An *i*-fetch copies data from a full shared variable to local object memory and leaves the shared variable intact. *S*-store and *i*-store operation do not cause the executing object to be suspended.

U-fetch and *u*-store operations on shared variables do not require access synchronization. A *u*-store operation can update the value of a shared variable unconditionally and a *u*-fetch operation copies the value from a shared variable to local memory unconditionally.

Successful extraction of a value from a full shared variable by a fetch operation (*x*-fetch or *s*-fetch) allows the first pending *x*-store request to resume or the first buffered value (*s*-store) to be stored. A successful store operation to an empty shared variable allows the first pending *x*-fetch, *s*-fetch, or all pending *i*-fetch requests to resume.

3 Overview of mEDA-2

mEDA-2 consists of two parts: the mEDA daemon and the mEDA library. The main function of the daemon is to manage parallel program termination. The mEDA library provides functions for accessing VSM.

VSM is a dynamic set of shared variables which are used for inter-task communication and synchronization. Data is stored in a shared variable in the form of a message packed in a PVM send buffer. The VSM is distributed among PVM tasks. Each variable is addressed by two components: (i) a task identifier, *tid*, and (ii) a variable identifier, *vid*. The user need not declare any shared variables; these are created dynamically by the store function and destroyed by the fetch-extract function.

The mEDA library provides the following functions to access VSM:

*eda_store(int op, int *tids, int n, int vid, char *m, int size)*

*eda_fetch(int op, int tid, int vid, char **m, int *size)*

*eda_prefetch(int op, int tid_from, int vid_from, int *tids, int n, int vid_to)*

In all functions, argument *op* defines one of four types of shared memory accesses: *EdaX*, *EdaS*, *EdaI* or *EdaU*, corresponding to *x*-, *s*-, *i*- or *u*-operations of EDA.

The function *eda_store* is used to store data to shared variables specified by the same vid and located in a number of tasks. The function *eda_fetch* is used to fetch data from a shared variable to local memory. The non-blocking function *eda_prefetch* generates a request to prefetch data from a shared variable to other shared variables with the same vid located in other tasks. A task can serve store, fetch or prefetch requests directed to it only when its computation is suspended.

The mEDA daemon insures the synchronous exit of all tasks, which are VSM users, to avoid deadlock while accessing shared variables which are distributed among the tasks. The termination problem is solved through exit barrier synchronization for VSM users.

4 Conclusions and Future Work

We have introduced mEDA-2, an extension of the PVM environment, to provide flexible and efficient mechanisms for inter-task communication and synchronization by means of Virtual Shared Memory and also to facilitate construction of parallel applications in PVM.

mEDA-2 was verified by implementing several parallel applications on a network of workstations. Our current plans include the implementation of mEDA-2 on real multiprocessors and the development of real-time mechanisms for it.

Acknowledgements

This research is supported by the Swedish National Board for Technical and Industrial Development, NUTEK (contract No. 93-3084). Vladimir Vlassov is holding a scholarship from the Wenner-Gren Center.

References

[1] O.A. McBryan, “An Overview of Message Passing Environments”, Parallel Computing, Vol. 20, No. 4, pp. 417-444, April 1994.

[2] A. Geist, et al., “PVM3 User’s Guide and Reference Manual”. ORNL/TM-12187, Oak Ridge National Lab. September 1994.

[3] H. Wu, “Extension of Data-Flow Principles for Multiprocessing”, TRITA-TCS-9004 (Ph D thesis), The Royal Institute of Technology (KTH), Stockholm, Sweden, 1990.

[4] J. Milewski, H. Wu, L.-E. Thorelli, “Specification of EDA0: An Extended Dataflow Actor model”, TRITA-TCS-EDA-9208-R, KTH 1992.

[5] H. Wu, J. Milewski, L.-E. Thorelli, “Sharing Data in an Actor Model”, Proc. 1992 Int. Conf. on Parallel and Distributed Systems, Taiwan, 245-250, 1992.

[6] H. Wu, L.-E. Thorelli, J. Milewski, “A Parallel Programming Model for Distributed Real-Time Computing”, Proc. Int. Workshop on Mechatronic Computer Systems for Perception and Action, Halmstad, 301-308,1993.

[7] L.-E. Thorelli, “The EDA Multiprocessing Model”. Technical Report TRITA-IT-R 94:28, CSLab, Dept. of Teleinformatics, KTH. 1994.

[8] V Vlassov, L-E Thorelli and H Ahmed, “Multi-EDA: A Programming Environment for Parallel Computations”. Technical Report TRITA-IT-R 94:29, CSLab, Dept. of Teleinformatics, KTH. 1994.