Implementation of the EARTH programming model on SMP clusters: a multi-threaded language and runtime system

(Extraits d'un article en préparation)

C.J. Morrone[1], J.N. Amaral[2], G. Tremblay[3], G.R. Gao[1]

[1] *Computer Architecture and Parallel Systems Laboratory (CAPSL), Dept. of Electrical and Computer Engineering, Univ. of Delaware, Newark, DE, USA*
[2] *Dept. of Computing Science, Univ. of Alberta, Edmonton, AB, Canada*
[3] *Dépt. d'informatique, Univ. du Québec à Montréal, Montréal, QC, Canada*

SUMMARY

We designed and implemented an EARTH (Efficient Architecture for Running THreads) runtime system for a multi-processor/multi-node, cluster. For portability, we built this runtime system on top of Pthreads under Linux. This implementation enables the overlapping of communication and computation on a cluster of Symmetric Multi-Processors (SMP), and lets the interruptions generated by the arrival of new data drive the system, rather than relying on network polling. We describe how our implementation of a multi-threading model on a multi-processor/multi-node system arranges the execution and the synchronization activities to make the best use of the resources available, and how the interaction between the local processing and the network activities are organized.

KEY WORDS: multi-threading, cluster computing, parallel programming language

*Correspondence to: José Nelson Amaral, Department of Computing Science, University of Alberta, Edmonton, AB, T6G 2E8, Canada

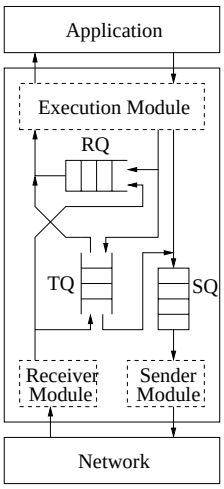

Figure 1. EARTH Runtime System

1. Introduction

This paper describes the design and implementation of the EARTH programming model on a cluster formed by symmetric multi-processor (SMP) nodes. This implementation requires an explicitly threaded language and a runtime system. The language, Threaded-C, has evolved from earlier developments of EARTH [6, 5]. In this paper, we discuss the new language features that we introduced in the version 2.0 of Threaded-C. Earlier versions of the runtime system were developed for other computing platforms [1]. Here, we describe the first completely functional implementation of the EARTH system on a cluster of symmetric multi-processor (SMP) nodes. This runtime system is designed for easy portability across Beowulf systems constructed with various processor nodes. It uses standard Unix sockets for inter-node communication and splits the tasks performed in the EARTH system — thread execution, communication, and synchronization — into three separate modules: an execution module, a sender module, and a receiver module. As discussed in Section 4, this organization of the runtime system is fundamental for an efficient and portable runtime system. This organization is also important to avoid deadlocks when using blocking I/O for interprocessor communication in the runtime system implementation.

This paper is organized as follows. Section 2 describes the EARTH programming model and its programming language, Threaded-C (release 2.0), the language used to write the programs run on the EARTH system. Section 3 describes the EARTH architecture model and the role of the runtime system (RTS). Section 4 discusses the new design of the RTS. Performance results (speedup curves) are then presented in Section 5, comparing our implementation of the EARTH runtime system for SMP clusters with an earlier implementation of such system. Results are presented for two clusters, one with 16 single processor nodes, and another with 64 dual-processor nodes. Finally, Section 6 presents related work.

2. The EARTH Programming Model and its Programming Language

Section omise (voir le manuel de référence de Threaded-C).

3. The EARTH Architecture Model and the Role of the Runtime System

Section omise (voir le manuel de référence de Threaded-C).

4. The Design of the New Runtime System

Our goal was to design an EARTH runtime system that is portable, makes efficient use of existing standard network interfaces, uses all the processing resources available in SMP Beowulf clusters, and delivers good performance.

The general structure of our new EARTH runtime system is shown in Figure 1. The Execution Module executes fibers and also takes on the responsibilities of intra-node scheduling, synchronization, communication and load balancing, tasks which were performed strictly by the SU in previous implementations of EARTH. The Receiver (resp. Sender) Module handles incoming (resp. outgoing) messages. The Token Queue (TQ) contains work that may either be performed by a processor within the local node, or that might be sent to a different node for execution. The Ready Queue (RQ) contains fibers that must be executed locally, and the Sender Queue (SQ) contains the outgoing messages.

In the following paragraphs, we describe the interface between the RTS and the network, and how our design for the RTS benefits from the resources available in an SMP machine. We also discuss the tradeoff between polling and interrupts, blocking I/O, potential deadlocks in an RTS, and the use of multiple processors in each node.

The case for standard Unix sockets

We chose the convenience of end-point communication provided by Unix sockets to establish the communication channels between multiple SMP processor nodes. Sockets provide an easy-to-use Application Programming Interface (API) that is consistent across many operating systems and networking hardware.

Earlier implementations of EARTH used custom network interfaces that allowed more efficient use of the network but were difficult to port to newer hardware and operating systems [1, 5].

To poll or not to poll

We know three alternatives to implement the communication between the runtime system and the network interface: interrupts, polling, and polling-watchdog. Interrupts are usually not desirable in a multi-threading system because they interrupt the running thread and lead to a context switch. The preferred method is for the runtime system to poll the network between the execution of threads, and thus avoid unnecessary context switching. A third method, *polling-watchdog*, developed especially for EARTH [2], mixes polling and interrupts in the following way: the runtime system polls the network between thread context switching, but when a message arrives, a timer starts counting; if the message is not handled within a given amount of time, the network interface interrupts the runtime system. The advantage of the polling-watchdog approach is that it prevents a thread containing a long running loop from making the node where it is running oblivious to what is happening in the remaining nodes of the cluster. To implement a polling-watchdog, however, it must be possible to program the network interface to obtain an appropriate time-out mechanism. This was done in earlier versions of the EARTH runtime system, but made those systems less portable.

Since Unix sockets are used for our inter-processing node communication, a polling approach would use the *select()* system call, requiring the kernel to perform a linear search on its socket structures to identify which socket has an incoming message. In a large cluster with many open sockets, polling can thus become expensive (between 2,500–15,000 processing cycles).

The drawback of interrupts is that they happen asynchronously with the thread context switching in a multi-threading system. Nevertheless, our decision to use Unix sockets makes interrupts unavoidable and care must be taken when handling them. When a message arrives, the Ethernet card raises a hardware interrupt that the CPU handles by stopping the currently running process. The OS interrupt handler decodes the interrupt and runs the appropriate hardware driver. When the driver is finished, the running process is allowed to continue at the point where it was interrupted. In our design, the kernel informs the runtime system of the arrival of a message and the runtime system immediately takes the actions required to process the message within the EARTH model before allowing the interrupted thread to resume execution.

Blocking vs. non-blocking I/O

After the runtime system has been notified that a message arrived, it needs to transfer the message's content, using a *read()* operation, from the socket buffer in kernel space into a buffer in user space. When the runtime system needs to send a message, it issues a *write()* operation to a socket. Both the read and the write operation behavior are affected by the use of blocking or non-blocking I/O. Such effect appears when a read requests more bytes than currently available in the socket buffer, or when the buffer overflows in the case of a write operation. In a non-blocking I/O system, any read or write operation will return immediately with the number of bytes that were successfully read/written. On the other hand, in a blocking I/O system, a call will not return (i.e., will block) until all bytes have been read/written.

Modern systems use default socket buffer sizes of 8192–61440 bytes [4]. Because modern processors are much faster than any available networking technology, these small buffers often become full. Thus the potential blocking situation can be frequent. If blocking I/O is used in a system where the same OS process handles both the network activities and the execution of EARTH threads, precious CPU cycles would be wasted when an I/O operation blocks. In our current design, we opted for a blocking I/O system but created two separate modules for the network activities. Thus, a blocked I/O operation does not block the processor indefinitely.

Avoiding deadlocks when using blocking I/O

When blocking I/O is used, a potential deadlock condition may arise when both the socket *send buffer* on one end of a link and the socket *receive buffer* on the other end of the same link become full. More precisely, both nodes might become blocked in a *write()* operation to the send buffer, both waiting for the other node to read from the corresponding receive buffer in order to allow communication to proceed. If both the writing and reading tasks are handled by a single process, then a deadlock situation will arise. [†] More general deadlock situations can also occur during the execution of multi-threaded programs that have complex cycles of inter-node dependencies.

Our solution to avoid such deadlocks is to create two separate modules in each processing node to handle network access: a sender module and a receiver module, as shown in Figure 1. Now, when the sender module blocks, the processor can switch to the receiver module and read any incoming data that is there and thus the system makes progress. If the scheduling is fair with respect to the receiver module, this decomposition into separate modules will then avoid deadlock situations, even if the execution module is allowed to proceed when the sender module blocks.

Benefiting from SMP

The RTS described here uses the blocking mode of access to sockets. We avoid polling, and use blocking calls to *select()*, which only returns when incoming traffic has arrived in a socket's receive buffer.

To avoid wasting cycles when blocking occurs — waiting for incoming network reads or waiting for outgoing network writes — the parts of the RTS that might block are decomposed into separate modules. Furthermore, these modules are distinct from the module responsible for running fibers.

The part of the RTS which runs fibers is called the *Execution Module* (EM). The two modules which, together, handle all networking activity, are known as the *Sender Module* (SM) and the *Receiver Module* (RM). These three modules are implemented as separate threads using POSIX threads (pthreads). The networking modules are only active when there is network traffic that needs to be handled; at all other times, they avoid wasting CPU cycles by going to sleep.

With the new modular design of the RTS, it becomes easier to provide SMP node support. Not only can the EM, SM, and RM run concurrently, but also multiple EMs can be implemented in the same processing node, with concurrent executions in multiple processors.

When multiple EMs are active, each has its respective Ready Queue, although they all share a Token Queue and a Send Queue. All of the modules are implemented as pthreads, and

[†] This situation did occur in an earlier implementation of the RTS, problem which was quite difficult to identify and fix since it was hard to reproduce because of the inherent time-dependent behavior in a parallel environment.

therefore share the same memory space. Intra-node communication is accomplished simply and efficiently through memory reads and writes.

5. Experimental Results

Section omise.

6. Related Work

Section omise.

7. Conclusion

This paper has presented the new design of the runtime system for the EARTH multithreaded architecture, together with the revised version of the Threaded-C language used to write programs for this architecture. The intended target machines for this new RTS are modern multi-nodes systems with multiple processors per node (SMP clusters). The RTS has been designed with the goal of being portable, yet making it possible to benefit efficiently from the power of multiple processors per node. In order to do this, the RTS implementation uses multiple threads of execution, which also precludes deadlock situations.

On the language side, current work is being done to further improve the Threaded-C language, yet preserve a narrow semantic gap that will ensure programmers still have full control over granularity of fibers and over synchronization and communication. For example, a notion of fiber with arguments has recently been introduced and experimented with (using a prototype pre-processor that extends the Threaded-C language [3]), allowing the flow of data to be made more explicit and simplifying further the specification of the synchronization constraints associated with fibers. Other extensions are still under investigation, for example, allowing fibers with multiple level of priorities.

ACKNOWLEDGEMENTS

The authors would like to thank current and former members of CAPSL at the University of Delaware for valuable ideas exchange. Special thanks to Kevin Theobald for the N-queens code, and to Juan del Cuvillo and Wellington Martins for the ATCG code. The authors also acknowledge the partial support from DARPA, NSA, NSF (under grants NSF-INT-9815742 and NSF-CSA-0073527), and NASA. The initial EARTH work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Guy Tremblay's work is also supported by an NSERC grant from Canada.

REFERENCES

1. P. Kakulavarapu, O. C. Maquelin, J. N. Amaral, and G. R. Gao. Dynamic load balancers for a multithreaded multiprocessor system. *Parallel Processing Letters*, 11(1):169–184, March 2001.
2. O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X.-M. Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *23rd Annual International Symposium on Computer Architecture*, pages 178–188.
3. J. Sauvageau. Extension du langage threaded-c: Fibres avec arguments et points d'entrée. Rapport de projet, bacc. en info. de gestion (prog. coop.), mai 2001.
4. W.R. Stevens. *UNIX Network Programming, Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Upper Saddle River, NJ, 1998.
5. K.B. Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.
6. K.B. Theobald, J.N. Amaral, G. Heber, O. Maquelin, X. Tang, and G.R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, University of Delaware, March 1998.