PEACE Threads Interface On Microkernel

Rajkumar
Venugopal K R
University Visvesvaraya College of Engineering,
Bangalore University, Bangalore, India

Mohan Ram N
Centre for Development of Advanced Computing,
Bangalore, India

Abstract - Recently, thread libraries have become powerful entities to support parallel programming on shared memory multiprocessors and multicomputers. However, the disparity between the primitives offered by the operating systems and thread's interface creates a challenge for those who wish to create portable threads library. The implementation of PEACE (**P**OSIX **E**xtensions to an **A**dvanced **C**omputing **E**nvironment) threads library conforms to the POSIX P1003.4a draft of threads interface standard on top of PARAS advanced computing environment on a PARAM supercomputer advented by C-DAC, India. The prime objective of this implementation is to support standard thread interface and enhance the portability of applications developed on PARAM, which was not possible earlier because of the usage of custom built thread interface (CORE), in the development of applications. PEACE threads provide portable threads within a single task, while allowing fully concurrent access to the system resources. Threads can be quickly created and synchronization among the multiple threads can be accomplished rapidly. These threads have the potential to liberate software developers from the limitations of the age old *fork()* model. In this paper, we discuss the parallel operating system model, microkernel, machine architecture, thread model, thread library architecture with implementation issues, and POSIX threads interface.

Introduction

The effectiveness of parallel computing depends on the performance of the primitives offered by the system to express parallelism. If the cost of creating and managing parallelism is high, even coarse grained parallel programs show poor performance. On the other hand, if the cost of creating and managing parallelism is low, even a fine-grained program can achieve excellent performance.

One way to express parallelism is by using UNIX-like processes with the age old *fork()* model [4]. Operating systems supporting such processes do not distinguish between a process and its address space (In this paper, the terms *task* and *process* are used interchangeably). Parallelism expressed using such heavy weight processes must be coarse grained, and are often not suitable for high performance parallel programming. Hence, in next-generation operating system kernels, address space and threads are decoupled so that a single address space can have multiple threads in execution [3]. Threads are an emerging model for expressing concurrency on multiprocessor and multicomputer systems. Thread is defined as a piece of code executing in concurrent with other thread. In multiprocessors, threads are primarily used to simultaneously utilize all the available processors, whereas in uniprocessor or multicomputer system, threads are used to utilize system resources effectively by exploiting the asynchronous behavior (opportunity for computation and communication overlap) of threads. As kernel-level threads offer a general programming interface to an application, they are expensive and hence, are not used in fine-grained parallel programs.

Unlike kernel-level threads, user-level threads are managed by runtime library routines linked to each application. Thread libraries enable application programs to use the thread management system which is most appropriate to the problem domain. Mach Cthreads [10], the University of Washington threads [11], SunOS LWP threads [8], C-DAC PARAS CORE threads [6], and POSIX Pthreads [9] are a few popular thread models.

Microkernel

The microkernel is aimed at migrating traditional operating system services out of the monolithic kernel into the user-level process. A microkernel is a tiny operating system core that provides the foundation for modular and portable extensions. Every next-generation OS will have one [7]. In

theory, a microkernel with a small privileged core surrounded by user-mode services, would deliver unprecedented modularity, flexibility, and portability.

The subsystems running on top of the microkernel implement their own kernels which can manage the resources better than the application ignorant general-purpose conventional OS mechanism [2]. The dominant representatives of this new generation OS technology are Mach, Chorus, Windows NT, QNX, and C-DAC PARAS.

Machine Architecture

The PARAM is an acronym for PARAllel Machine. The PARAM family of supercomputers include PARAM 8000, PARAM 8600, and PARAM 9000. The PARAM 8600 is a distributed memory, message passing parallel computer [1] based on the i860 RISC processor and works as a back-end Compute Engine to hosts such as PC/AT, SUN workstations, Micro VAX machines, and U6000 machines. It provides multi-user facility by partitioning back-end compute nodes into one or more logically disjoint fragments and assigning them to users.

The parallel programming environment for C-DAC PARAM is known as PARAS. At the lowest level, the parallel programming model offered by the PARAS microkernel is one of kernel-level threads. PARAS offers constructs for tasks, threads, memory management services, and synchronous/asynchronous communication between the threads of different tasks or the same task [5]. PARAS microkernel is a message passing operating system kernel designed for massively parallel systems. The kernel is replicated on each node of the system supporting multiple tasks with a paged virtual memory space, multiple threads of execution within each task, and message based interprocess communication.

Thread Model

Parallel programs have multiple threads of control. On a multiprocessor system, each thread runs on an individual processor. An application level scheduler schedules threads on top of kernel supported threads, which in turn are scheduled by a kernel scheduler on the available processors. The computational model is shown in Figure 1, where kernel-level entities form a cluster of virtual processors for user level threads. It results in true parallelism if the number of threads and physical processors map is 1:1, otherwise, they are scheduled on virtual processors.

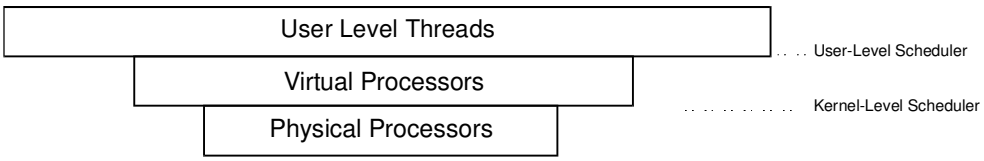

Figure 1 Computational Model

The thread structure defines a thread of control. Thread is started by giving it a start-procedure to execute and a set of arguments for that procedure. The change in the state of the thread from ready to run, provides an opportunity for the thread to acquire the CPU and start its execution. From time to time, a thread may block waiting for some event, and another thread will be scheduled. When the thread's start-procedure terminates, the thread is removed from the service by reclaiming the resources. A task initially has a single thread, running the *main(..)* procedure.

A task with a single thread of control is equivalent of an UNIX-process. Programming a task having multiple threads of control is known as multithreading. Threads may be viewed as execution resources that may be applied to solving problems. Threads share a process instructions and most of its data. A change in shared data by one thread can be seen by the other threads of the same task.

There are a variety of synchronization facilities to allow threads to cooperate in accessing shared data. Each thread has a program counter (PC, instruction counter), a stack to maintain local variables, and return address.

Cost of Thread Execution

Thread based parallel programs spend a significant amount of time in dynamic operations such as creating/deleting threads, allocating/deallocating memory segments etc. The total cost of a parallel application can be computed as follows**:**

$$Ca = \alpha * (Csync) + \beta * (Ccsw) + \gamma * (Cdyn) + Ci$$

where Ca, Csync, Ccsw, Cdyn, Ci are the **a**pplication cost, **sync**hronization cost, **c**ontext **sw**itching cost, **dyn**amic operation cost, and other thread **i**ndependent cost respectively. The parameters α, β, and γ increase monotonously (due to increase in search space of threads data structure) with the number of threads. We posit that Ci remaining constant, the minimum value of Csync, Ccsw, and Cdyn produces the minimum value for Ca. Further, these parameters are bounded by hardware speed and kernel primitives.

Thread Library Architecture on PARAM

Thread library provides the programmer, interface for multithreading. They are implemented by utilizing the underlying microkernel supported threads of control. The architecture of PEACE thread library is shown in Figure 2.

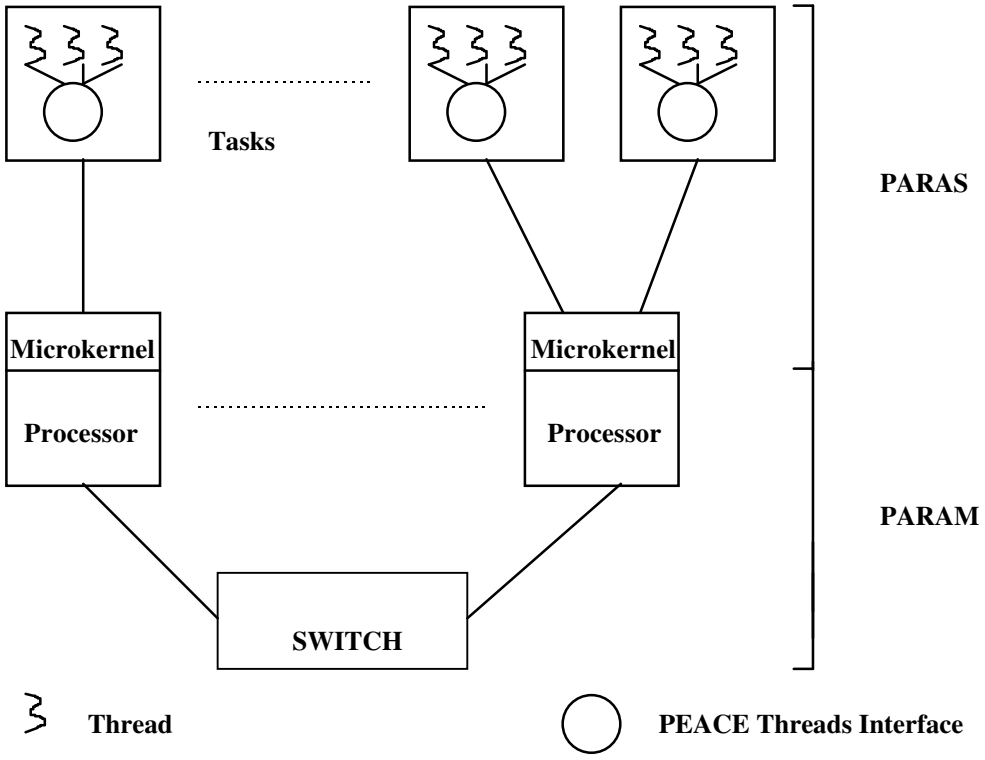

Figure 2 PEACE Threads Interface Architecture

A user task creates multiple threads using thread-library calls. The PARAS microkernel in co-ordination with PEACE threads interface schedules threads based on *scheduling factors* on a single processor. The microkernel suspends the execution of threads for scheduling high priority threads. PARAS microkernel thread scheduler, uses Round-Robin preemptive algorithm for scheduling of threads. The thread runs until time-out, or blocks waiting for some event, or deliberately gives up the CPU to another thread in the case it completes its execution within a time-slice allocated to it.

Each thread is represented by a thread structure that contains a thread ID (identification), an area to store the thread execution context, thread mask, thread priority, and pointer to a thread stack. The storage for the stack is either automatically allocated by the library or is passed in by the application while creating a thread.

PEACE threads interface is not designed to protect users from each other or to arbitrate access to shared resources, but it merely provides runtime support to a single application program running within PARAS on PARAM. It is entirely user-level code, and does not require kernel modification to run on any of the PARAM family of supercomputers platform. A consequence of this implementation is that the system is easy to modify or extend to support the specified need of advanced users. PEACE threads interface is implemented in C language using microkernel system calls.

POSIX threads Interface

The interface calls supported by PEACE thread library of POSIX P1003.4a Pthreads draft are grouped into the following categories:

- Thread Creation and Control
- Thread Scheduling
- Thread Synchronization
- Thread Cancellation and Cleanup
- Threads and Signals

Managing Multiple Threads: Management of threads involves thread creation, running them independently or as a group, suspending and resuming them as and when required.

Thread Creation: The POSIX threads creation and control interface calls lack in terms of features for expressing real parallelism on multiprocessor or multicomputer systems. POSIX threads once created, start executing, which does not allow thread group parallelism. This is overcome in PEACE threads package by the *pthread_par* interface call, which schedules all the specified threads (threads group) for parallel execution. The thread creating process continues only after the thread group has finished execution.

Controlling threads: Once a thread is created, it continues to execute until it completes. However, it can be suspended/resumed using the interface calls *pthread_suspend*/*pthread_resume*. The creating thread can wait for completion of the thread in execution state using *pthread_wait* call.

Threads Synchronization:

In PEACE, thread synchronization is supported by semaphore mechanism. A semaphore is a protected memory location that tells one thread that another is currently using a shared resource, and therefore it should wait. The user applications must ensure that a process can read the semaphore, modify the semaphore, and write the semaphore back to a protected memory location without the interference of another task accessing the semaphore during the read-modify-write sequence. PEACE threads support mutex (binary semaphore) and counting semaphore.

The semaphore mechanism is implemented on the i860 processor locking mechanism with *lock* and *unlock* instructions. The atomic instructions are built upon by using *lock* and *unlock* pair to handle shared resources with protected memory location. The software layer of synchronization interface calls are built on top of these atomic instructions. They include, *pthread_mutex_destroy, pthread_cond_init, pthread_cond_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread _cond_wait, pthread_cond_timewait, pthread_cond_signal*, and *pthread_cond_broadcast*.

Parallel Programming Support

A sequential program is one which runs on a single processor and has a single line of control. To make many processors collectively work on a single program, the program must be divided into smaller independent chunks so that the processors can work on separate chunks of the problem. The program decomposed in this way is a parallel program. PEACE threads library facilitates the usage of process parallelism and data parallelism approaches in the parallelisation of the program. This can be exploited by application to express parallelism.

PEACE threads services ensures that these subprograms work in a coordinated manner. Programs can be split into maximum number of subprograms without really bothering about the number of processors required to execute them.

Comparison with Other Thread Models

Comparison between various other implementations of threads interfaces with PEACE is discussed below.

Mach C Threads: It is a runtime subsystem developed by CMU, but applications developed using this interface lack portability.

Chorus Threads: Chorus intentionally avoids user-level threads because of a perceived impact on real-time requirements.

University of Washington Threads: This is a portable thread interface, but does not support programmer control over the kernel resources.

CORE threads: It is a runtime subsystem developed by C-DAC on top of PARAS on PARAM, but applications developed using this interface lack portability.

Sun LWP threads: It is a portable thread interface, but has no explicit kernel support. It suffers from application blocking when a LWP calls a blocking system call or a page fault takes place.

PEACE Threads: PEACE thread library is a classic user-level thread package, which conforms to POSIX P1003.4a Pthreads draft. Parallel applications developed using PEACE threads can easily be ported to other platforms, such as Windows-NT, OS/2, or Chorus as they support POSIX standard.

Conclusions

The PEACE threads interface calls offer the following advantages:

- Allow the application developer to decouple logical program parallelism from the relatively complex kernel supported parallelism in terms of application program interface.
- Programmer can control the allocation of stack for threads local-storage requirements.
- Portable and competitive to other threads interface model.

So far in PARAM supercomputer, only a custom-built threads interface CORE was available. With the availability of POSIX threads Interface and standard message passing interfaces like Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) on PARAM under PARAS, the portability of applications developed under PARAS is extended.

Acknowledgement

The authors are grateful to all the members of C-DAC, Bangalore, and in particular to the members of Operating System Group, Mr. Venkatakrishnan, Mr. Bala Kishore, Mr. Hari Prakash, Mr. Achutharaman, Mr. Thirumalai, Mr. Balaji, Mr. Rama Rao, and Miss. Jacqueline for their useful conversations and comments during the implementation and an earlier draft of the paper.

References

1. C-DAC, *Advanced Computing*, Tata Mc-Graw Hill Publications, 1991.
2. David and Kenneth, *A caching model of operating system kernel functionality*, operating system review, ACM Press, Jan. 1995.
3. Budhisattwa, Grey and Kaushik, *A Machine independent interface for lightweight threads*, operating system review, ACM press, Jan. 1994.
4. Dr. Dobb's Journal, *Thread Programming in UnixWare 2.0*, June 1995,
5. C-DAC, *PARAS Microkernel interface manual for PARAS 8600,* February, 1994.
6. C-DAC, *Concurrent Runtime Environment (CORE) for PARAS 8600,* February 1994.
7. BYTE, *Advanced Operating Systems*, Jan. 1994.
8. Steven and Associates, *Writing Multi-threaded Code in Solaris*, Sun Micro Systems Inc., 1992.
9. POSIX P1003.4a IEEE Draft, *Threads Extension for portable operating system.*
10. E. Cooper and R. Draves, C Threads, *Technical Report CMU-CS*, Department of CSE, Carnegie Mellon University, June 1988.

11. Mc Jones and Swart, *Evolving the UNIX Systems Interface to Support Multithreaded Programs*, International proceedings on USENIX winter conference, 1988.
12. Edward and Dylan, *Improving the Performance of Message Passing Applications by Multithreading*, Proceedings of Scalable High Performance Computing Conference,
Copyright © IEEE, 1992.

About the Authors

Rajkumar is a distinguished graduate in Computer Science and Engineering from the University of Mysore. He was awarded Gold Medal for securing the I Rank in the year 1992. He is currently a Research Scholar and is pursuing Post graduate studies at UVCE, Bangalore University, Bangalore. He has published a book, Microprocessor x86 Programming. He is a Member of Technical Staff, Operating System Group, C-DAC, Bangalore. His research interests include Graphics, Software Tools Development, Microkernel OS for MPP, and Parallel Algorithms.

Venugopal K R is serving on the faculty of Computer Science and Engineering at University Visvesvaraya College of Engineering, Bangalore University, Bangalore. He has published numerous books on Computer Science including the best sellers Petrodollar and the World Economy and Microprocessor x86 Programming. He is listed in Who is Who in the World and 5000 Personalities of the World. His research interests include Parallel Processing and Distributed Computing.

Mohan Ram N is Group Co-ordinator, Operating System and Networking, Centre for Development of Advanced Computing, Bangalore. He has completed Master of Technology in Computer Science from the Indian Institute of Technology, Madras, in the year 1985. He is with C-DAC, since its inception and his research interests include Parallel Architecture, Operating System, and Networking.