Systems and Computers in Japan, Vol. 29, No. 2, 1998
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J80-D-I, No. 1, January 1997, pp. 42–49

Design and Implementation of a Parallel Pthread Library (PPL) with Parallelism and Portability

Chikara Sakamoto, Teruki Miyazaki, and Masayuki Kuwayama

Faculty of Engineering, Kyushu University, Fukuoka, Japan 812

Keizo Saisho and Akira Fukuda

Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Japan 630-01

SUMMARY

Although multi thread libraries have been implemented to provide threads—a unit of concurrent/parallel execution—at user level, there is no thread library that provides both parallelism and portability. We designed and implemented the PPL (Parallel Pthread Library) with the following two requirements in mind: (1) It should permit parallel execution according to the system, and (2) it should provide a common thread environment for many operating systems and be highly portable. We employed a multi-routine approach to realize parallelism, separating the internal structure of PPL into two parts to expedite portability: namely, a virtual processor dependent module and a virtual processor-independent module. We implemented PPL in several systems and compared its performance with other user-level threads. Furthermore, we evaluated parallelism in multi-processor systems. The results show that PPL has sufficient portability and parallelism. © 1998 Scripta Technica. Syst Comp Jpn, 29(2): 28–35, 1998

Key words: User-level thread; thread library; parallelism; portability; performance evaluation.

1. Introduction

Since multi-processor systems began to be used in business, opportunities for general use of multi-processor systems have been growing. Recently, workstations with about 5 to 15 processors have been employed. This trend is spreading to the world of personal computers, suggesting that it is indispensable to generalize a programming environment that has parallelism and portability.

Usually, each UNIX process is assigned a process in order to execute multiple processes on multi-processors cooperatively and concurrently. However, since each UNIX process has its unique virtual space, the overhead of process switching is large. Thus, the control flow (called a thread or light-weight-process) is extracted from a process and used as the unit of processor assignment. A thread represents the execution path and consists of a program counter, a stack pointer, and a stack. The thread has the advantage that the overhead for its generation and termination can be kept small compared with a process, and it can perform synchronization and communication rapidly and easily.

Threads are classified into kernel level threads and user level threads depending on the implementation method [1]. Kernel level threads accompany system calls and their overhead is larger than user level threads. Therefore, it is generally believed that user level threads are more efficient. However, threads are executed sequentially, not concurrently, because many current user level threads are implemented as coroutines. Furthermore, many thread systems employ proprietary thread interfaces, so that it is difficult to port applications that use threads. Thus, there is no thread library that has both parallelism and portability which will become indispensable features in the near future.

Thus, we designed a thread library PPL (Parallel Pthread Library) with two requirements in mind: (1) capability for parallel execution according to the system, (2) common thread environment on many operating systems. In order to satisfy (1), we implemented user level threads as multi-routines that can be executed on multiple virtual processors provided by OS. To satisfy (2), we explicitly distinguished a virtual processor dependent part and an independent part, and standardized the interface between them.

We implemented PPL on BSD UNIX, System V UNIX, and Mach 2.5, and verified its high portability. Furthermore, we confirmed that parallelism in a system can be derived from experiments on the multi-processor systems implemented.

We describe thread control approaches and their problems in section 2 of this article. In section 3, we show the design strategy of PPL. We describe the implementation of PPL in section 4 and evaluate it in section 5.

2. Thread Control Approaches

We will describe the control approaches of kernel level threads and user level threads. Also, we will state problems in the control approaches of existing user level threads.

2.1. Kernel control approach

Threads provided by the OS kernel are called kernel level threads. Kernel level threads are implemented in OSs such as Mach [3], that are used for multi-processor systems and distributed environment. Since each thread is controlled by the OS kernel, a thread is the unit of processor assignment (Fig. 1). Therefore, it has the advantage that threads can be scheduled so as to utilize system parallelism efficiently. On the other hand, the kernel takes part in every operation such as thread generation and thread termination. Thus, it is necessary to switch spaces from the user space to the kernel space and to guard resources from unauthorized operations. This creates the disadvantage that thread operation overhead is large.

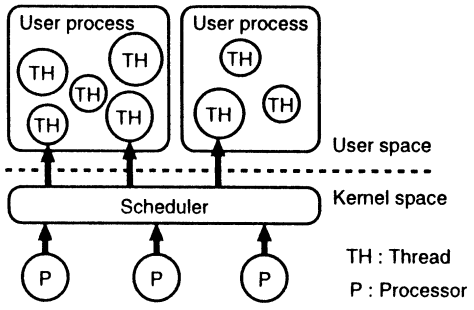


Fig. 1. Kernel control approach.

2.2. User lever control approach

A thread controlled not by the OS kernel but at the user level is called a user level thread. User level thread control approaches are classified as coroutine approaches and multi-routine approaches.

The advantage of the user level control approach is its light weight. Control overhead can be kept very small, because every thread operation, such as thread generation and thread termination, is performed at the user level.

2.2.1. Coroutine approach

In a coroutine approach, threads are implemented as coroutines within a process (Fig. 2). A coroutine is executed by sequential transfer of control among multiple procedures and operations are provided as a library or language environment. The advantage of this approach is that it is easy to implement. However, since each thread is implemented as a coroutine, only sequential execution can be

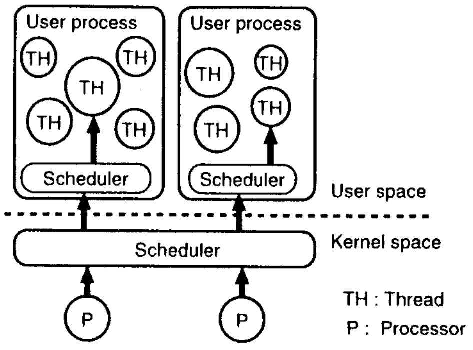


Fig. 2. Coroutine approach.

performed. One example of a coroutine is the LWP (Light-Weight-Process) in SunOS [4].

2.2.2. Multi-routine approach

In a multi-routine approach, user level threads are executed on multiple virtual processors provided by the OS and the threads are actually executed concurrently (Fig. 3). A virtual processor is a process or a kernel thread provided by the OS as the unit of processor assignment on multi-processor systems. The coroutine approach can be seen as a multi-routine approach with only one virtual processor.

User level threads can be executed in parallel by the multi-routine approach and a thread system with light weight and parallelism can be achieved. However, it is more complex then the coroutine approach, because integrity of thread control management data must be retained among virtual processors.

Mach's Cthreads feature is implemented by the multi-routine approach, where Mach's kernel level threads are used as virtual processors [5].

2.2.3. Problems with conventional approaches

Many user level thread systems have been provided as libraries. Among public thread libraries are Mach's Cthreads, the SunOS thread library developed by Muller at University of Florida [6] (here called FPL), and the Portable Thread Library (PTL) developed by Abe and coworkers at Osaka University [7].

Cthreads employs the multi-routine approach with the Mach thread as a virtual processor and it has parallelism. However, since it heavily depends on Mach threads, it is difficult to transport it to other OSs.

FPL realizes a fast thread library by limiting its target to SunOS (Sparc). It does not have parallelism or portability.

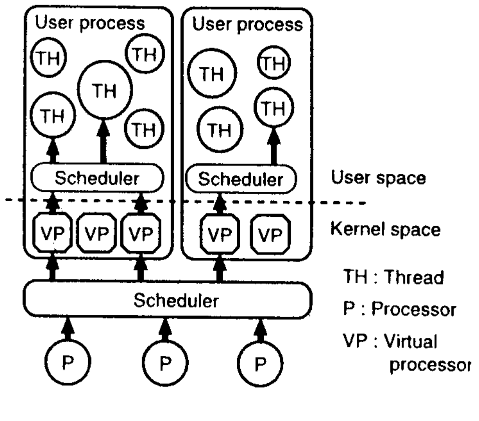


Fig. 3. Multiroutine approach.

PTL's target is BSD UNIX and it can be executed on other UNIXes with little modification. Thus, its portability is excellent. Also, it has many features such as automatic stack extension, non-blocking I/O, and debugging functions. However, threads are realized as coroutines within a UNIX process and it does not provide parallelism.

There is no thread system that has both parallelism and portability, which will be indispensable features in the future. Thus, we are developing PPL to create a library that has both parallelism and portability.

3. Design of PPL

In this section, we describe the design strategy, configuration, and implementation of PPL.

3.1. Design strategy of PPL

We designed PPL with two objectives: (1) it can utilize the parallelism of a system; and (2) it provides a thread interface that is common to many OSs.

3.1.1. Realization of parallelism

We realized parallelism by using the multi-routine approach. Although the virtual processor model provided by each OS is unique, we addressed this problem.

3.1.2. Realization of portability

In considering portability, we should take into account compatibility with other thread libraries and portability of the thread library itself.

User interface

We used the Pthread interface [9] as PPL's thread interface in order to provide a common interface to users.

The Pthread interface is a standard thread interface established by IEEE as POSIX (POSIX 1003.4a Thread Extension). Since it is expected that many OSs will provide thread functionality with the Pthread interface, we employed the Pthread interface for PPL to increase application portability.

Portability of thread library

One of problems in library portability is that the virtual processors provided by systems differ with the OS. For instance, in Mach a virtual processor is given as a Mach thread, and the interface for its generation and termination is unique.

PPL's internal structure is explicitly separated into a virtual processor dependent part (VP dependent module) which deals with the interface with virtual processors, and a virtual processor independent part (VP independent module) which provides the Pthread interface. By using this separation, the necessary modifications for library portability are limited, portability is easily understood, and the portability of PPL is increased (Fig. 4).

VP interface

The VP interface is composed of the part that performs virtual processor operation and the part that controls exclusion.

The part which performs virtual processor operation:

vp_create()	to generate a virtual processor.
vp_terminate()	to terminate the specified virtual processor.
vp_suspend()	to suspend the virtual processor that issued this function.
vp_resume()	to wake up a suspended virtual processor, and to re-execute the thread.

The part which controls exclusion by spin locks:

spin_init()	to initialize a lock.
spin_lock()	to obtain a lock. to wait until a lock is obtained.
spin_try_lock()	to obtain a lock. returns a failure code when no lock is obtained.
spin_unlock()	to release a lock.

Exclusion within the library

Exclusion control for data structures within the library, such as ready queues, is needed in parallel execution on multi-processor systems and the unit of exclusion control is a problem. In order to improve parallelism, some measure is needed, such as partitioning a critical section using a lock variable for each data structure. In systems where deadlock control or the number of lock variables is limited, other means must be used.

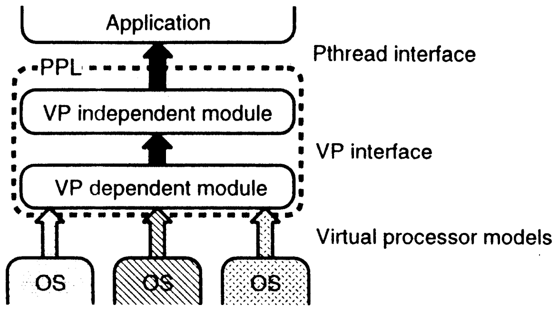


Fig. 4. Structure of PPL.

Currently in PPL, one lock variable is used for exclusion control of the whole library, so that it can be ported to an OS which has only one lock variable available for lock primitives. In future, it will be necessary to develop a mechanism that optimizes exclusion control according to the number of lock variables.

3.2. Structure of PPL

3.2.1. Virtual processor dependent module

The virtual processor dependent module absorbs the differences of virtual processors on different OSs and presents a unified virtual processor interface to the virtual processor independent module.

There are two ways to generate virtual processors: one is to generate a fixed number of processors, and the other is to dynamically generate/terminate processors. Generation and termination of virtual processors is performed in the kernel, so the overhead is greater than user level threads. Therefore, a fixed number of virtual processors is generated at library initialization time in PPL, and all virtual processors are terminated at library termination time.

However, in future, it will be necessary to consider dynamic generation and termination of virtual processors according to the degree of parallelism of applications, and assignment of virtual processors according to the number of applications.

3.2.2. Virtual processor independent module

In the virtual processor independent module, user level threads provide the Pthread interface and are realized as multi-routines that run in parallel on the virtual multiprocessor system provided by the virtual processor dependent module.

In the following, we describe points that influence portability and implementation methods to increase portability.

Context switching

Context switching is usually described in the Assembler language, since it deals with storing/recovering processor registers. It is largely dependent on the system creating a problem for transportation.

In PPL, context switching is done using C language standard functions, e.g., *setjmp* and *longjmp* [8].

Stack reservation

Each thread is assigned an independent stack. Four methods to reserve a stack for threads are described in [7].

Table 1. Specification of target machines

	Sun IPX	LUNA88K	CHALLENGE
Processor	Sparc $\times$ 1	MC88100 $\times$ 4	MIPS R4400 MC $\times$ 36
OS	SunOS 4.1.3	Mach 2.5	IRIX Release 5.2
Virtual processor	UNIX process	Mach thread	UNIX process

(1) Static Data Area: to reserve a stack statically within the data area.

(2) Heap Area: to reserve a stack within the heap area; this is superior to (1) in that a stack of any size can be reserved.

(3) Shared Memory: to use shared memory segments as stack; this has the advantage that stack overflow can be detected.

(4) Redzone Protect Stack: to detect stack overflow by using the *mprotect* system call; stacks can be reserved in the heap area.

PPL employs shared memory as the stack area, because it is used by almost all recent UNIXes and can detect stack overflow.

Setting stack pointers

The stack pointer (SP) is set so that it points to the stack assigned to a generated thread. There are three methods to set the SP.

(1) Direct Setting: to directly set the SP register. It has little portability, because it is necessary to describe it in Assembler.

(2) Indirect Setting: to use *setjmp* and *longjmp*; standard functions of UNIX. The current context is stored in *jmp_buf* by *setjmp*, and a place used to store SP within *jmp_buf* is rewritten. It can be implemented using C language. It is highly portable when the structure of *jmp_buf* on the target system is known.

(3) Setting by Signal Stack: This is used in PTL. It uses the signal stack mechanism provided by BSD UNIX after and including 4.2BSD. The signal stack mechanism specifies a stack for signal handler execution. This method is commonly used in OSs with this mechanism and portability is very high.

PPL uses method (3) for improvement of portability. However, PPL uses method (2) when the signal stack mechanism is not available.

4. Implementation of PPL

We implemented PPL on a Sun IPX, an OMRON LUNA88K and the multi-processor system SGI CHALLENGE. Table 1 shows the number of processors, OS and virtual processors of each system. We will describe two major problems that occurred while implementing PPL.

Restriction of the number of shared memory segments

Shared memory segments have the following restrictions, and the number of threads that can be provided is restricted when shared segments are used as stacks:

- the size of one shared memory segment.
- the number of shared memory segments one system can have.
- the number of shared memory segments that can be assigned to one process.

For the above reasons, we shared one memory segment among multiple threads when multiple threads are used. This method cannot detect stack overflow; nor can the heap method, but we verified by experiments that thread switching is faster than the heap method.

Stack check at *setjmp*

This problem occurred when implementing context switching on, LUNA88K (OS: Mach2.5). In the LUNA88K, the restored SP is checked at *longjmp* if it points to a lower address than current SP and, if so, error termination occurs. That is to protect against unexpected behavior caused by the stored SP pointing to a used stack. In PPL, we implemented context switching by rewriting *longjmp* so that it does not check SP.

5. Evaluation

We evaluated the basic functions of PPL by comparing them with other thread libraries. Also, we evaluated

Table 2. Execution time for basic thread operations (μs)

	Sun IPX			LUNA88K			CHALLENGE
	PPL	PTL	FPL	PPL	PTL	Cthreads	PPL
Context switching	130	85	35	180	350	50	35
Thread generation	1400	1800	1700	3000	2600	6500	1200

parallelism by executing parallel programs on multi-processor systems.

5.1. Evaluation of lightness

In order to evaluate the lightness of threads provided by PPL, we compared the thread operation performance of PPL with that of other thread libraries. We used PTL and FPL on a Sun IPX, and PTL and Cthreads on a LUNA88K for comparison. There is no thread library to be compared on the CHALLENGE, so we show only the performance results.

Comparison items

We compared context switching and thread generation as thread operations.

- context switching

We generated two threads and measured the time taken to perform context switching between them 1,000 times. We used one virtual processor in order to measure the time for context switching alone.

- thread generation

We measured the time taken to generate 100 threads.

Table 2 shows the execution results for these two test programs. The measured time is real, and we performed measurements in a way that they were not affected by other users.
Table 2 shows the following.

- The speed of context switching is inferior to FPL and Cthread, which perform switching by Assembler, but at the same level as PTL, which performs switching by *setjmp* and *longjmp*.
- The speed of context switching on CHALLENGE is reasonable considering the relative performance of the processors used by CHALLENGE and SunIPX.
- The speed of thread generation is at the same level as other libraries.

5.2. Evaluation of parallelism

We evaluated parallelism on the multi-processor system CHALLENGE by comparing the time it takes to execute a parallel program with one virtual processor and the time when multiple processors are used.

Test program

We used matrix multiplications as a test program. This program generates N threads in $N \times N$ multiplications and each thread computes one row (with N elements). Figure 5 shows the flow of threads. We performed 200×200 matrix multiplications and measured sequential execution time and parallel execution time. Figure 6 shows the result.
Figure 6 shows the following.

- Sequential execution time did not increase much even if the number of virtual processors increased. This is because thread generation was performed by one virtual processor. We improve PPL, and evaluate tradeoffs between parallelism and lock

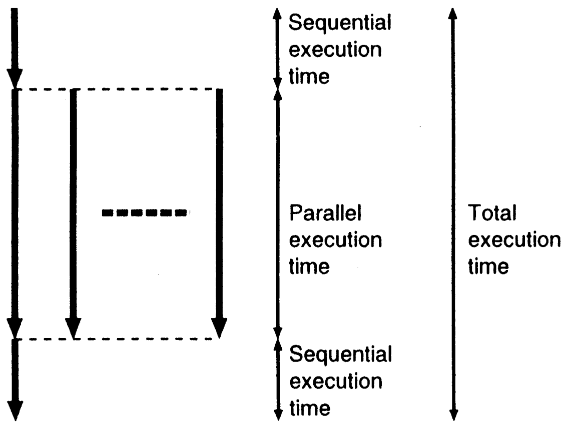


Fig. 5. Execution flow of threads in a matrix multiplication.

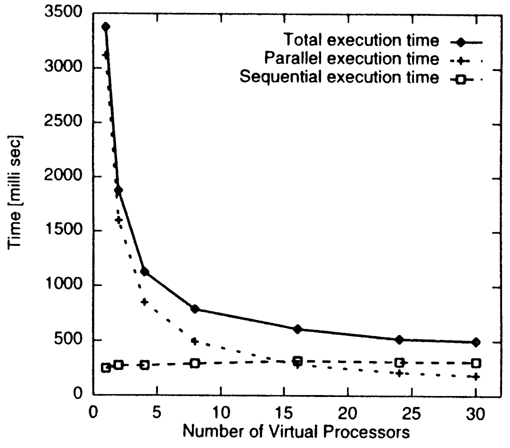


Fig. 6. Execution time for matrix multiplications.

overhead when virtual processors are generated in parallel.

- The effect of parallelism is explicit enough for parallel execution time. This shows that PPL has sufficient performance for problems having the granularity of this test program.

5.3. Evaluation of portability

It is difficult to evaluate portability because there is no explicit evaluation standard.

We implemented PPL first on the Sun IPX (OS: SunOS 4.1.3) and ported it to the LUNA88K (OS: Mach 2.5) and to CHALLENGE (OS:IRIX Release 5.2). About 20% of the virtual processor dependent module was modified when porting to LUNA88K, and about 15% in porting to CHALLENGE. Those values are by no means small, but we think they are permissible in porting.

Modification of the virtual processor independent module was necessary in cases such as *longjmp* on LUNA88K. This means that separation criteria for virtual processor dependent and independent modules must be studied further in the future. Furthermore, it is necessary to determine whether the two layer structure is sufficient or not.

6. Conclusions

We have described PPL, a user level thread library that we are studying and developing in order to clarify problems in existing thread systems, and to satisfy requirements needed to solve those problems, with parallelism and portability in mind.

In PPL, we implemented user level threads as multi-routines that are executed in parallel on virtual processors provided by the OS as the processor assignment unit in various systems. PPL provides threads that have both the light weight of user level threads and the parallelism of kernel threads.

In order to improve portability, we separated the internal structure of PPL into a virtual processor dependent module and an independent module. It was expected that due to this separation, modifications during porting would be confined to the virtual processor dependent module. However, it was necessary to modify part of the independent module in the actual porting that we performed. This means that the two layer modularization we designed it is not sufficient to absorb differences between systems. Thus, in the future, it may be necessary to review the modularization and to introduce an intermediate module.

Furthermore, we compared the lightness of PPL with other user level thread libraries. The result showed that the overhead of thread operations is at the same level as other user level libraries. It is necessary to make threads faster in the future.

On the multi-processor system CHALLENGE, we evaluated parallelism using a test program. The result showed that PPL has sufficient performance for parallel processing when the granularity is medium.

Among future tasks are:

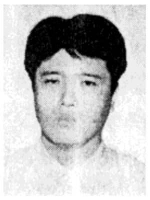
- porting to other multi-processor systems.
- porting to various other OSs.
- performance evaluation using real applications with different types of parallelism.
- study of dynamic virtual processor generation algorithms.
- implementation of the Pthread interface and its study.
- review of modularization.

REFERENCES

1. A. Fukuda. Parallel operating system. Information Processing, **34**, No. 9, pp. 1139–1149 (1993).
2. T. Miyazaki et al. Parallel pthread library (PPL): user-level thread library with parallelism and portability. Proc. COMPSAC'94, pp. 301–306.
3. M. Accetta et al. Mach: a new kernel foundation for unix development. Proc. the Summer 1986 USENIX Technical Conf., pp. 93–112.
4. Sun Microsystems. SunOS Reference Manual. (1988).
5. C. Cooper and P. Draves. Cthreads. Technical Report CMU-CS-88-154, Carnegie Melon University (1988).

6. F. Mueller. A library implementation of POSIX threads under UNIX. Proc. the Winter 1993 USENIX Technical Conf., pp. 29–41.
7. H. Abe, T. Matsuura, and K. Taniguchi. An implementation of light-weight process mechanism with high portability on BSD UNIX. Articles of Information Processing, **36**, No. 2, pp. 296–303 (1995).
8. Y. Tada and M. Terada. An implementation method of C coroutine library with high portability and extensibility. Shingaku-Ron (D-I), **J73-D-I**, No. 12, pp. 961–970 (Dec. 1990).
9. IEEE. Threads extension for portable operating systems. P1003.4a/D6 (1992).

AUTHORS (from left to right)



Chikara Sakamoto received his B.S. and M.S. degrees from Kyushu University, Information Engineering, in 1993 and 1995, respectively. Joined Kyushu Denko in 1995. Interested in parallel processing in general and system software.

Teruki Miyazaki received his B.S. and M.S. degrees from Kyushu University, Information Engineering, in 1992 and 1994, respectively. Joined Nippon Steel in 1994. Interested in parallel processing in general and system software.

Masayuki Kuwayama received his B.S. degree from Kyushu University, Information Engineering, in 1991. Entered Ph.D. program at Kyushu University. Interested in operating systems and networks.

Keizo Saisho (member) received his B.S. and M.S. degrees from Kyushu University, Information Engineering, in 1982 and 1984, respectively. Research associate, lecturer, and assistant professor of Kyushu University, Information Engineering in 1984, 1991, and 1993, respectively. Assistant professor of Nara Institute of Science and Technology, Information Science since 1994. Ph.D. Engaged in the study of high reliability systems, parallel processing, and concurrent processing. Member of the Information Processing Society.

Akira Fukuda (member) received his B.S. and M.S. degrees from Kyushu University, Information Engineering, in 1977 and 1979, respectively. Joined NTT Laboratories in 1979. Research associate and assistant professor at Kyushu University, Department of Science and Engineering, Information Systems in 1983 and 1989, respectively. Professor at Nara Institute of Science and Technology, Information Science since 1994. Ph.D. Engaged in studies of operating systems, parallelizing compilers, computer architecture, parallel/distributed processing, and performance evaluation. Received research award and best author award from Information Processing Society in 1990 and 1993, respectively. Cotranslator of "Concepts of Operating Systems." Member of ACM, IEEE Computer Society, Information Processing Society, and Japan OR Society.