1993

Efficient Implementation of Thread Migration on Distributed-Memory Multi-processors

Janche Sang

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Report Number:
93-065

EFFICIENT IMPLEMENTATION OF THREAD MIGRATION ON DISTRIBUTED-MEMORY MULTIPROCESSORS

Janche Sang
Vernon Rego

CSD-TR-93-065
October 1993

Efficient Implementation of Thread Migration on Distributed-Memory Multiprocessors *

Janche Sang
Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

October 1993

Abstract

Executing multi-threaded programs on multiprocessors can exploit the inherent parallelism among threads. However, the load imbalance and frequent remote data access factors may degrade the performance in distributed-memory parallel systems. To cope with these problems, one of the solutions is to enhance threads with dynamic migration capability. This paper describes our experiences with the design and implementation issues of thread migration in the Xthreads library, which is currently running on the nCUBE2 and iPSC860 machines. Performance measurements of the current implementation are also included.

* Research supported in part by PRF, ONR contract 93-1-0233, ARO contract 93-G-0045, and NATO award 900108.

1 Introduction

Lightweight processes or *threads* have emerged as a representation of computational entities, cooperating with each other within a process to achieve a common goal. Spreading execution of threads over several processors can exploit parallelism and thus achieve better performance. However, in distributed-memory parallel systems, two factors may degrade the performance gains from multi-threading. The first factor is load imbalance. For example, during program execution, there may be a dense cluster of threads resident on a single processor even though only few threads exist on other processors. Thus, lightly loaded processors have to wait for heavily loaded processors to complete their work. The second factor is non-local data access. Threads will typically access non-local data and thus require unavoidable inter-processor communication. Consequently, these two factors bring the need to provide the threads with dynamic migration capability.

With the migration capability, a systematic scattering of threads across processors which allows heavily loaded processors to efficiently balance their load with lightly loaded processors gives the executing system an opportunity to achieve a better overall throughput. Also, if cross-processor data access is going to be frequent, then relocating the accessing thread to the site hosting the remote information can reduce inter-processor communication traffic.

The above two reasons motivate us to design and implement the thread migration primitive on a thread library called *Xthreads*[8], which was designed to enhance the programming language C with concurrent capability in the form of library functions and sets of predefined data structures. The purpose of the Xthreads library is to support a cheap concurrent programming environment. To the best of our knowledge, no thread package supports thread migration function on these machines.

Our goal in this work is to provide a migration primitive to tune program organization for high performance computing, such as numerical computation, simulations, etc.; therefore, efficiency is a major concern in our design. Furthermore, in order to take full advantage of scalable parallel architectures, the scheme of migration has to scale well for thousands of processors. In other words, running the software for large numbers of processors should allow it to take advantage of the increased computing capacity. Finally, the interface should be simple and straightforward, but sophisticated enough to meet the needs of a wide range of

applications.

The remainder of the paper is organized as follows. Section 2 details the design rationale and related work. Section 3 introduces the migration interface. Section 4 discusses the implementation in more detail. Early performance measurements are given in Section 5. A brief conclusion is presented in Section 6.

2 Design Rationale

The Xthreads library supports logical *concurrency* within each processor and physical *parallelism* across processors in distributed-memory systems. Two-level logical concurrency is provided by using *heavyweight* processes and *lightweight* processes (i.e. threads). Heavyweight processes begin with the same program image and then initialize the Xthreads environment. Multiple threads of control sharing a single address space are created within a process. Physical parallelism is realized through distributing and executing these processes across processors. In general, an Xthreads program consists of a set of processes and threads, cooperating and communicating through the interprocessor network.

Figure 1 depicts the structure of the Xthreads model. Each processor can have more than one process running concurrently. In the Xthreads library, a process is the logical unit to host threads. That is, threads can be created and migrated dynamically across process and/or processor boundaries.

We allow more than one process to run at each processor because the Xthreads library is built at the user-level. The user-level threads suffer performance losses in that when a thread invokes a system call or encounters a page fault, the whole process has to be blocked[6]. This blocking effect can be attenuated by using non-blocking system calls. However, not all system calls provide a non-blocking version. Two-level concurrency can remedy this problem. By increasing the degree of multiprogramming on a processor, a processor can choose other processes to run when the running process is blocked. Hence, performance can be improved because the waiting time is overlapped with computation.

The semantics of the thread migration function is that a migrant thread will resume its execution at the statement following its point of migration, as if nothing occurs. The behavior of migration is like the manner of conducting a context switch, except that migration resumes

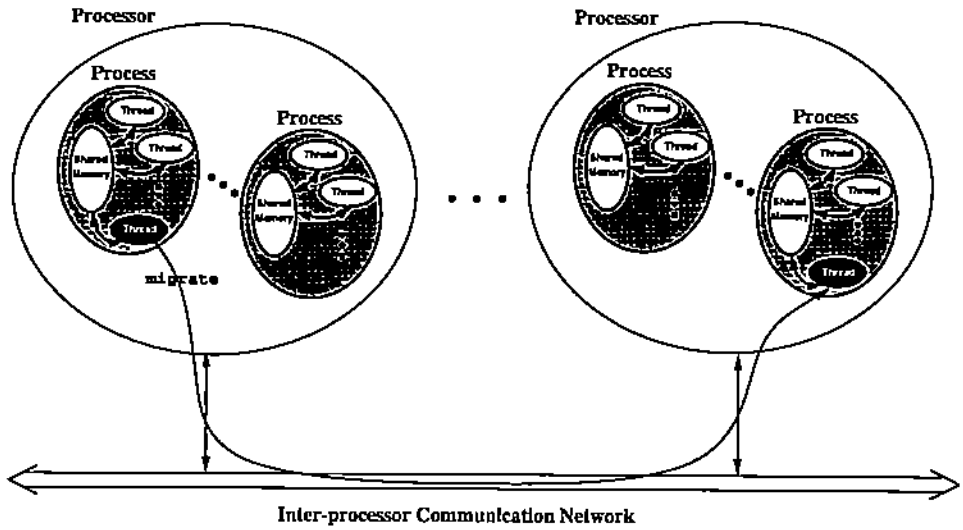


Figure 1: The Enhanced Xthreads Programming Model

the execution of the thread on a different process. Therefore, a thread's context must be transported from one process to another in a way that allows it to resume execution on the latter at a point of suspension on the former. Figure 2 compares the difference between the context switching mechanism and the migration mechanism. As shown in part (b), the migration mechanism is realized cooperatively by the *source* process and the *destination* process. The source process is responsible for suspending a thread, gathering the thread's state, such as the control block information and the runtime stack, into a message, and transferring the message to the destination process. If the migrant thread is the running thread, the scheduler of the source process will select the next thread to run after sending the message. The destination process decomposes the migrant thread's state message, inserts it into the pool of ready threads, and resumes it at an appropriate time according to the scheduling policy.

2.1 Related Work

Many process migration mechanisms have been developed in the past few decades[10]. Recent distributed systems with process migration capability include Charlotte[1] and Sprite[4]. However, as compared with thread migration, process migration is considerably more expensive because process context is larger than the context of threads.

Recent multithreaded systems supporting thread migration capabilities include IVY[5] and Amber[3]. The IVY system was designed at the kernel level to support shared virtual memory.

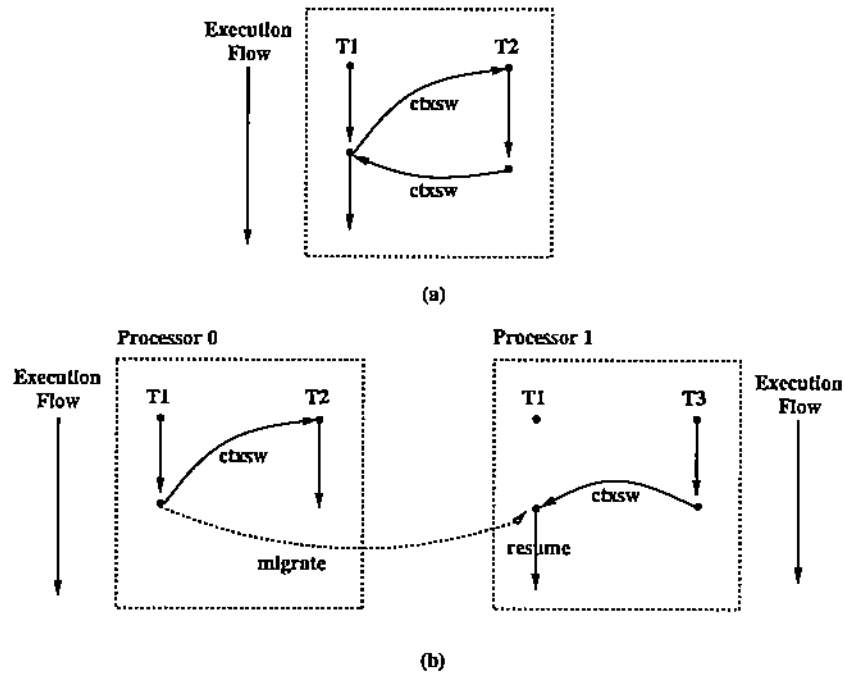


Figure 2: Thread Context Switch vs. Thread Migration

However, it requires the support of special purpose hardware to detect a page fault. The Amber system integrates several shared-memory multiprocessors (DEC Firefly[11]) through Ethernet and always moves threads for the remote data accessing purpose. Our thread migration mechanism differs from Amber in that it is designed to achieve load balancing and/or to increase data locality. Furthermore, our system allows a thread to be moved to another process, even if the process is in the same processor.

3 The Function Interface

The calling interface of the new function is described below:

```
XTCALL xthread_migrate(xptr,nid,pid)
xthread_t xptr;
int nid;
int pid;
```

Three parameters are needed to invoke the function `xthread_migrate()`. The first parameter `xptr` is a pointer which refers to the migrant thread. The second parameter and the third

parameter, i.e., the destination node identifier `nid` and the process identifier `pid`, uniquely determine the location of the process which will host the migrant thread.

This simple interface not only allows a thread to move some other threads, but also allows a thread to move itself. For example, a thread can use the following function invocation

```
xthread_migrate(xthread_self(),nid,pid);
```

to move its execution to the remote process `pid` at node `nid`.

4 Implementation Issues

The enhanced Xthreads library with thread migration capability has been implemented on the nCUBE2 and iPSC860 distributed-memory multiprocessors. We chose these two machines because of their availability in our research environment, their popularity in a wide variety of parallel machine users, and their continued development. Moreover, good performance of passing messages in inter-processor communication networks, which are three or four orders of magnitudes cheaper than Ethernet, makes the migration primitive more efficient and thus more attractive to programmers.

4.1 The Function xthread_migrate()

As mentioned, a thread can invoke the function `xthread_migrate()` to move itself or other threads to another process. Moving a thread which is not the running thread is not difficult, because its context has been saved when the thread is suspended. However, if a running thread is willing to migrate itself, the migration function has to store the thread's current state, especially the position at the top of the stack and the value of the program counter. We propose two approaches to implementing the migration function. The first approach is written in assembly code, while the second one is built on top of the context switch function. Both are described below.

Approach I: written in assembly code

An ideal implementation is to let the migrant thread resume its execution at the statement immediately following the invocation of the function `xthread_migrate()`. To achieve this, we have to save the return address to the caller of `xthread_migrate()`. According to the

nCUBE's calling conventions, the return address is stored on the stack which is indicated by the stack pointer SP. Unfortunately, we cannot write an asm statement in a C function to access the stack pointer since the first instruction generated by the C compiler for each function is to decrement the stack pointer. This adjustment is designed to reserve some space on the stack for storing actual parameters and local variables, but it also destroys the original value in the stack pointer. Therefore, we resort to using assembly code to implement the migration function.

Figure 3 shows the code of the function `xthread_migrate()` in nCUBE2 assembly language. This function does not have a pair of code to decrement the stack pointer at the beginning and to increment it back before returning as a normal function does. Once the function is invoked, it first saves the stack pointer if it finds the caller thread trying to do a self-migration. Next, after saving a few registers which are needed for resuming later, it calls a high level function `_xsendstate()`, as shown in Figure 4. Both function `xthread_migrate()` and `_xsendstate()` have the same formal arguments. So the function `xthread_migrate()` just bypasses these arguments through registers without actually pushing them again onto the stack. The function `_xsendstate()` first composes the state information into a message, then transfers the message to the destination process, and finally kills the migrant thread since it no longer exists in the source process.

Figure 5 depicts the stack when a thread invokes the function `xthread_migrate()`. Only the lower part of the stack (from the bottom to the place indicated by stack pointer) needs to be transferred. When the migrant thread resumes through a context switch, the return address on the stack is popped up and is then loaded into the program counter register. Therefore, execution continues immediately after the statement which invokes the function `xthread_migrate()`.

4.2 Approach II: built on top of the context switch function

The disadvantage of Approach I is its usage of low level assembly language which is hard to understand and maintain. If we allow the self-migration thread to resume execution inside the function `xthread_migrate()` instead of returning directly to the caller, the migration mechanism could be implemented on top of the context switch function `ctxsw()` and hence the whole function could be written in C. Of course, we should ensure that resuming in the

```
        .file   "migrate.s"
        .code

_xthread_migrate:
        ! register r0: xptr, register r1: nid, register r2: pid
        .pub    _xthread_migrate
        cmpw    r0, __currxt            ! if xptr == _currxt, save SP
        bne     $no.save
        stpr    #16, 4(r0)              ! store stack pointer to xregs[SP]
                                        ! NOTE: offset 4 is xregs position

        ... saving registers to current thread's buffer...

$no.save:
        call    _sendstates             ! if xptr == _currxt, never return
        ret
        .elftype        _xthread_migrate, ?function
        .size   _xthread_migrate, .-_xthread_migrate

        .ext    _sendstates
        .lang   ?Clang
```

Figure 3: The function xthread_migrate() on nCUBE2

```
_xsendstate(xptr,nid,pid)
{
    • compose the thread xptr migration states, including stacks,
      registers, and control information, into a message;
    • send the message to the process pid in the remote node nid;
    xthread_destroy(xptr);
    /* no return if xptr is the running thread */
}
```

Figure 4: The private function invoked by xthread_migrate()

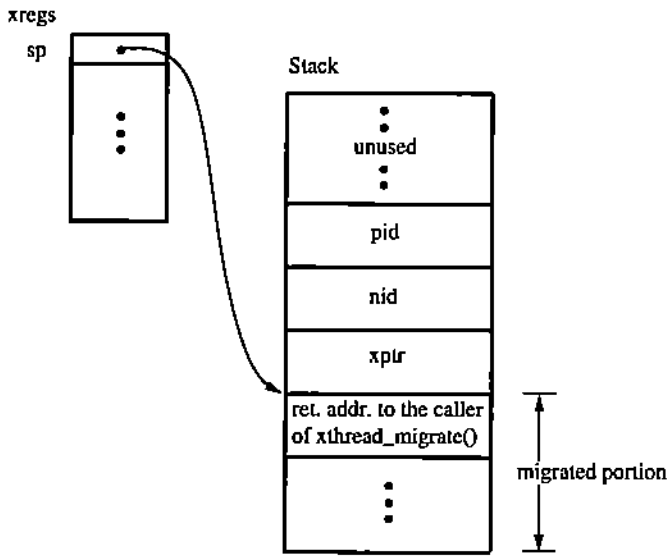


Figure 5: The Stack Frame after Invoking `xthread_migrate()`

middle of `xthread_migrate()` will behave the same as the immediate return.

Figure 6 and Figure 7 show the pseudo-code of `xthread_migrate()` and its corresponding stack frame after invocation, respectively. If it is a self-migrating case, the migration function calls the function `ctxsw()` by using the current thread's register buffer as the two arguments. The execution of `ctxsw()` will continue while it has left a copy of the context in the thread's buffer. The migration message is then composed and sent to the remote process after evaluating the condition

$$(\texttt{nid} \,!= \texttt{\_nid} \;||\; \texttt{pid} \,!= \texttt{\_pid}). \tag{1}$$

which is true the first time. The global variables `_nid` and `_pid` represent the processor identifier and process identifier, respectively. Because of the invocation of the function `ctxsw()`, the migrant thread resumes the execution at the statement following the invocation of `ctxsw()`. In addition, the function `ctxsw()` records the useful stack area (see Figure 7), including the three actual parameters `xptr`, `nid` and `pid`. So these parameters are still with the same value in the destination process. Thus, the migrant thread will re-evaluate Condition (1), which yields false this time because of a different pair of `_nid` and `_pid`. Finally, control returns to the caller of `xthread_migrate()`.

The disadvantage of this approach is its inefficiency because of the extra condition eval-

uation, two function returns to return to the caller, and the larger size of the stack to be migrated.

The above two approaches differ only in the thread self-migration case. For the case of moving a suspended thread, these two approaches perform almost in the same way. The suspended and migrant thread resumes its execution after the suspension point.

4.3 Data Moving

Consider the data copying tasks taken in the composition and decomposition of a migration message. A naive implementation is to allocate a large enough buffer and then copy the migrant thread's control information, register values, stack, etc. together into the buffer. The reverse procedure does in a similar way in the destination site. A buffer with the same size is used to store the corresponding message and is then decomposed into different parts. Each part, such as control block and stack, is copied to the migrant thread's corresponding areas accordingly (see Figure 8(a)). Hence, a large amount of work in data moving would definitely increase the migration cost, especially when the stack size is not small.

The cost of data copying can be reduced by using the migrant thread's stack as the message buffer. We know that the area whose address is smaller than the value in SP is unused. Therefore, the source process only needs to move the control information to this unused space (i.e. on top of the stack) to form the message. Furthermore, the destination site can abstract the control information from the front of the message since the size of a thread control block is fixed (see Figure 8(b)).

4.4 The Destination Site

The main task in the destination site is to decompose the migration message and insert the migrant thread into the ready list. Figure 9 shows the pseudo-code for handling the migration message. Since the message size is known when it arrives, we can calculate the starting address, i.e. upper, of a stack area to receive the message. After moving the control block information which is in the upper part of the stack, we adjust the stack pointer to point to the top of the stack (see Figure 8(b)). The migrant thread is ready to run when it is inserted into the ready list.

```
xthread_migrate(xptr,nid,pid)
{
    if( xptr == _currxt)
        ctxsw(xptr->xregs,xptr->regs);
    /* the migrated thread will resume execution here and */
    /* evaluate the following conditions again.  */
    if(nid != _nid || pid != _pid) {
        • compose the thread xptr migration states, including stacks,
          registers, and control information, into a message;
        • send the message to the process pid in the remote node nid;
        xthread_destroy(xptr);
        /* no return if xptr is the running thread */
    }
}
```

Figure 6: An alternative implementation of xthread_migrate()

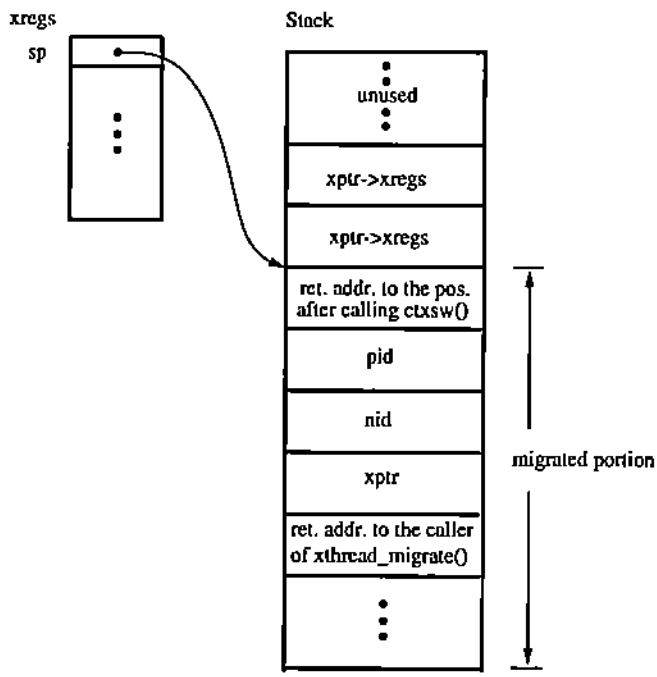
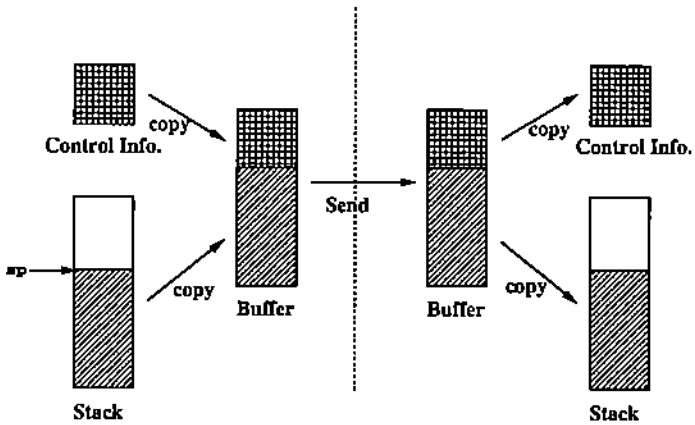
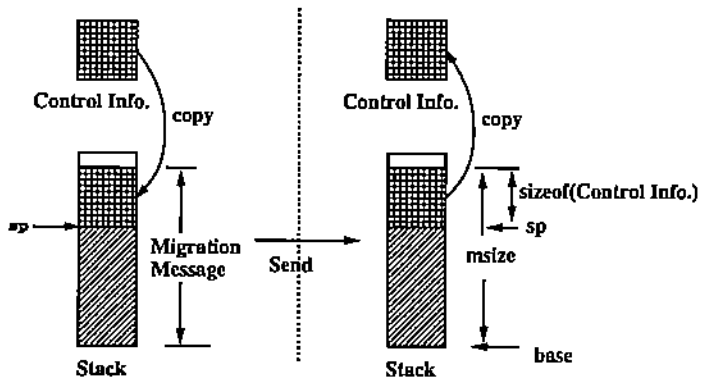


Figure 7: The Stack Frame after Invoking xthread_migrate() in Approach II



(a) Using Explicit Buffer



(b) Using Stack as Message Buffer

Figure 8: Using Stack Area As Message Buffer

```
comm_server()
{
    ...
    if((msize=ntest(&source,&type)) >= 0) { /* there is a message */
        switch(type) {
        case MIGRMSG:
            • find a free thread entry from thread table and let xptr
              point to it;
            upper = xptr->xbase - msize + sizeof(WORD);
            nread(upper,msize,&source,&type,&flag);
            • move context from the stack top the thread entry.
              xptr->xregs[SP] = upper + sizeof(struct CONTEXT);
            • insert xptr to the ready list.
            break;
        ...
        }
    }
    ...
}
```

Figure 9: Processing the Migration Message

4.5 The Global Address Space

In the Xthreads environment, since processors are homogeneous and all processes execute the same image, each thread has the same view of the address space. The addresses of functions and static global data are identical to all threads, regardless of where the threads reside. Therefore, translation for these static addresses can be avoided because of the common address domain. However, dynamic allocation of data (eg. using malloc()) during program execution may destroy the global view. Furthermore, a pointer which refers to the location on a stack will become undefined if the stack is migrated but not stored in the same area. Since the current version Xthreads library does not solve these problem completely. programmers have to restrain themselves from using these capabilities.

Our ongoing implementation is to modify the strategy used in [3] which separates the global heap space and assigns a distinct segment to each node. Unfortunately, this approach wastes most of the memory. For example, 99.9% heap space will be unused for each processor in a 1024-node parallel machine. Furthermore, the scalability of the system is greatly limited. Since the heap space is also used for thread stacks, the number of threads cannot be scaled

up linearly with the number of processors. We modify this strategy by allowing two or more threads to use the same stack area. If collision occurs, only the running thread can use the stack area, while the other suspended threads' stacks are swapped out in temporary space.

Since there is no perfect implementation which can meet all of our objectives, we provide compiler-time options for users to build the Xthreads library. If efficiency is the major concern and no dynamic structure is used, the old version is enough to meet the programmers' need. Otherwise, the alternate implementation can be selected.

4.6 Experiences on iPSC860

Because of some machine dependent features, a few modifications are needed to implement the migration function on iPSC860. For example, the return address is saved in the register *r1*, instead of being pushed on the stack as on the nCUBE2. More register values also have to be saved and transferred because of compiler optimization.

5 Performance Measurement

In this section we present some early performance measurements for the thread migration latency. To evaluate the migration cost, we created a thread which travels back and forth on two processors for a large number of times and then measured the average of these. Table 1 shows the performance comparison of the null thread migration costs using Approach I and II discussed in the previous section. Approach II needs to revaluate a condition twice and also requires one more function return. Therefore, it is not surprising to see that it yields slightly larger overheads than Approach I.

We carried out an experiment to see the impact of the stack size on the performance of thread migration. Table 2 shows the results by varying the size of local data (in bytes) defined inside a thread. The more local data used in a thread, the larger the stack that is needed, the longer transmission overhead there is, and therefore a higher migration cost is incurred. For the purpose of comparison, we measured the performance of the Naive approach which uses an extra buffer to transmit and receive migration messages, and the Stack/Buffer approach which uses the migrant thread's stack area as the message buffer. The results in Table 2 show that the Stack/Buffer approach outperforms the Naive approach, because the former need not to copy the stack at all, whereas the latter may need to copy the stack twice. The performance

Approach	I	II
Migration Cost	253	263

Table 1: Null Thread Migration Cost (in μs) using Approach I & II

Local data (in bytes)	16	64	256	1024
Naive (on nCUBE2)	311	437	909	2805
Stack/Buffer (on nCUBE2)	259	299	403	838
Naive (on iPSC860)	158	180	355	686
Stack/Buffer (on iPSC860)	144	156	290	459

Table 2: Migration Latency (in μs) with Different Size of Local Data

gains also become larger when the stack size increases. From the results in Table 2, we can see that the performance of the Stack/Buffer approach on nCUBE2 improves 15% when the local data size is 16 bytes, and even up to 70% if the data size is 1024 bytes.

Table 3 shows a detailed cost breakdown for thread migration using the Stack/Buffer approach on the nCUBE2 machine. This table is produced from a combination of timing measurements. Since we have reduced the stack copying overhead, the overhead for sending a message dominates the execution time. The copying overhead in the table involves marshaling and unmarshaling the control information, register values, etc. to and from the stack. The thread destruction cost is not included in the table because the cost can be overlapped with the message sending time in the source process.

We also measured the message transmission time between two connected processors. The results for both nCUBE and iPSC860 machines are shown in Figure 10. Each of the communication delays has a constant startup latency. The delay grows linearly with the message length on the nCUBE2, while it has a jump at the 100-byte message length on the iPSC860. More communication properties of these two machines can be found in [7] and [2]. This feature favoring short messages on iPSC860 gives us an alternative way to transmit the migration message. If both control information and stack are small, we can send them separately instead of merging them together. The performance can be improved because the total cost of sending two small messages could be less than the cost of sending a combined message which is over

Category	Overhead	Percent
Message transmit	241	81%
Copy data	16	5%
Get a thread entry	7	2%
Insert to ready list	10	3%
Context Switch (yield)	12	4%
Dequeue from ready list	8	3%
Others	5	2%
Total Time	299	100%

Table 3: Breakdown of Time (in μs) for Migration (64-byte local data)

100 bytes. Furthermore, data moving cost can be reduced totally because of the avoidance of message composition and decomposition.

6 Conclusion

This paper has presented practical and efficient implementations of thread migration, a mechanism that suspends running of threads on a processor and resumes execution on another processor. Performance measurements show that the migration cost is close to the minimum message transferring overhead. The cheap migration primitive gives programmers the opportunity to tune system performance by balancing loads across processors and by increasing data locality.

We also faced implementation tradeoffs between efficiency and high-level programming. Two approaches were proposed to implement the migration function. One is to use assembly code to save the register values for later immediate return. The other is built on top of the context switch function, which enjoys the benefit of high-level design but with slightly higher overheads. Furthermore, our experiences suggest that the thread's stack can be used as the message buffer to avoid redundant data moving.

Several applications have been realized using the enhanced Xthreads library with the thread migration capability as the basic software support to parallel computing on the nUCBE2 and the iPSC860 machines. Relevant results have been achieved in the field of the parallel-distributed simulation applications. A novel mobile-process approach has been proposed to parallelize a process-oriented simulation system[9].

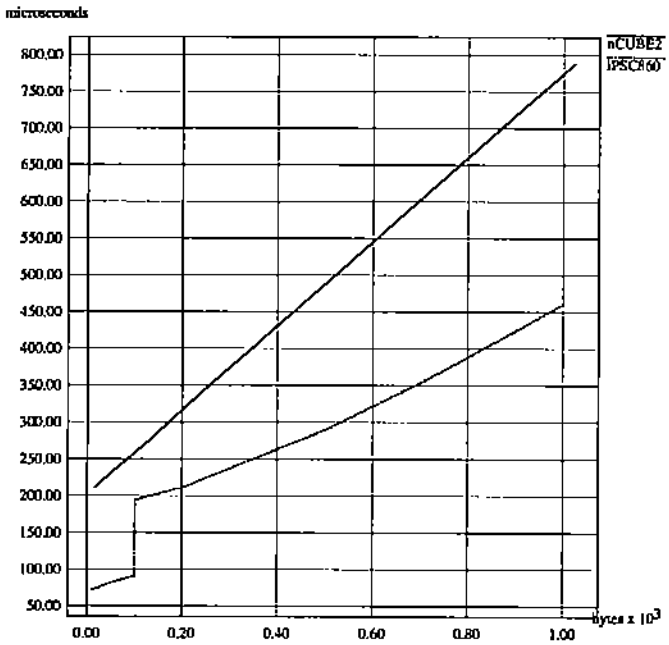


Figure 10: Message Transmission Latency in nCUBE2 and iPSC860

References

[1] Y. Arsty and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, pages 47–56, Sep. 1989.

[2] R. Berrendorf and J. Helin. Evaluating the Basic Performance of the iPSC/860 Parallel Computer. *Concurrency: Practice and Experience*, pages 223–240, Sep. 1992.

[3] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Symposium on Operating System Principles*, pages 147–158, 1989.

[4] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21:757–785, Aug. 1991.

[5] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pages 147–158, 1988.

[6] B. M. March, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class User-level Threads. In *Symposium on Operating System Principles*, 1991.

[7] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Network. *IEEE Computer*, 26:62–76, Feb. 1993.

[8] J. Sang, F. Knop, V. Rego, J. K. Lee, and C.-T. King. The Xthreads Library: Design, Implementation, and Applications. In *Proceedings of the COMPSAC*, 1993.

[9] J. Sang, E. Mascarenhas, and V. Rego. Process Mobility in Distributed-memory Simulation Systems. In *Proceedings of the Winter Simulation Conference*, 1993.

[10] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating System Review*, pages 28–40, July 1988.

[11] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite Jr. Firefly: a Multiprocessor Workstation. *IEEE Trans. on Computers*, 37:909–920, Aug. 1988.