Vol.2012-ARC-202 No.8
Vol.2012-HPC-137 No.8
2012/12/13

Porting MassiveThreads Thread Library to FX10 Supercomputer System

Nan Dun[1,a)] Jun Nakashima[1,b)] Kenjiro Taura[1,c)]

Abstract: We ported MassiveThreads, a light-weight thread library, to FX10 supercomputer system. We illustrate the technical challenges of implementing context switching for SPARC architecture, and identify the performance bottlenecks in context switch implementations in current Linux kernel and GNU C library. The evaluation shows that our user-level implementation in assembly is over 30x faster than using library context switch routines, e. g., ucontext series. Our implementation also enables the Chapel programming language to run more efficiently on FX10 when using MassiveThreads than other tasking layers such as Pthreads, Nanos++, and Qthreads.

1. Introduction

Modern computers are equipped with CPUs having more and more cores, which provides a promising future of next-generation computer systems but also proposes significant challenges to software systems to harness the power of such massive parallelism.

First, the number of cores for one CPU is increasing and each core is usually assigned with 10∼100 threads during the execution time. This requires fine-grained multithreading and approaches to hide the latency from memory and network. Secondly, connections between nodes are becoming faster thus I/O should be efficiently performed to avoid its limit on overall performance.

To this end, we designed and developed the MassiveThreads, a middleware for writing high performance applications [10], [11]. MassiveThreads is a light weight thread library designed for handling I/O efficiently and scalable execution of multiple threads. It is designed to achieve following goals: 1) efficient management of a large number of threads, especially with small overhead to create and destroy them; 2) efficient execution of I/O in many-node/core environments; 3) interface that is compatible to existing programs without compilation.

The PRIMEHPC FX10 supercomputer system [8] located in The University of Tokyo, manufactured by Fujitsu Limited, is a massively parallel supercomputer with a peak performance of 1.13 PFLOPS. It consists of 4,800 computing nodes (16 cores per node, i. e., 76800 cores in total) with SPARC64 IXfx processors based on SPARC64 V9 architecture [14]. Computing nodes of FX10 are connected via the 6-dimensional mesh/torus Tofu interconnect [1] to achieve high scalability and fault tolerance.

Therefore, FX10 is an ideal target platform of MassiveThreads and we made an effort to enable MassiveThreads to run on FX10. Accordingly, the contribution of this work is as follows:

- We ported MassiveThreads thread library to FX10 by providing an efficient implementation in assembly of context switching for SPARC V9 architecture. Evaluation shows that our implementation is over 30x faster than the implementation using library context switch routines and achieves good scalability up to 16 cores in one single node.
- We also identified the overheads of current context switch implementation in GNU C library and Linux kernel, which illustrates that our implementation is generic and applicable to improve the implementation of existing library and kernel implementation.
- We further extend our effort to enable Chapel programming language [13] to efficiently run on FX10 with MassiveThreads as a tasking layer. In addition, experimental comparisons show that MassiveThreads tasking layer achieves better performance and scalability than other tasking layers such as Pthreads, Nanos++ [5], and Qthread [15].

The rest of this paper is organized as follows. In Section 2, we briefly describe the SPARC architecture and corresponding challenges imposed to the implementation of context switching. The details of our implementation and related analysis is presented in Section 3. Section 4 shows the evaluation results of our implementation comparing to several reference implementations on FX10. We finally summarize our work in Section 5.

2. The SPARC Architecture

The SPARC is a RISC (Reduced Instruction Set Computing) instruction set based architecture. Its most popular revisions include the 32-bit SPARC V8 and 64-bit SPARC V9. The details of these architectures can be found in SPARC reference manuals [12], [14]. Besides instruction set, we briefly introduce two SPARC major particularities related to context switching when comparing to other architectures (e. g., x86 and AMD64): the register windows and stack layout.

[1] The University of Tokyo, 7-3-1 Hongo, Tokyo 113–8656, Japan
[a)] dun@eidos.ic.i.u-tokyo.ac.jp
[b)] nakashima@eidos.ic.i.u-tokyo.ac.jp
[c)] tau@eidos.ic.i.u-tokyo.ac.jp

Table 1 Integer Registers of SPARC Architecture

Registers	Alias	Descrptions
Global Registers		
%r0	%g0	zero, no effect for writes
%r1	%g1	temporary value
%r2	%g2	reserved by libraries/compiler
%r3	%g3	reserved by libraries/compiler
%r4	%g4	reserved by libraries/compiler
%r5	%g5	reserved for SPARC ABI
%r6	%g6	reserved for SPARC ABI
%r7	%g7	Thread local storage
Output Registers		
%r8	%o0	out parameter 0 / in return value
%r9	%o1	out parameter 1 / in return value
%r10	%o2	out parameter 2 / in return value
%r11	%o3	out parameter 3 / in return value
%r12	%o4	out parameter 4
%r13	%o5	out parameter 5
%r14	%o6 / %sp	stack pointer
%r15	%o7	address of CALL instruction
Local Registers		
%r16	%l0	local value 0
%r17	%l1	local value 1
%r18	%l2	local value 2
%r19	%l3	local value 3
%r20	%l4	local value 4
%r21	%l5	local value 5
%r22	%l6	lcoal value 6
%r23	%l7	local value 7
Input Registers		
%r24	%i0	in parameter 0 / out return value
%r25	%i1	in parameter 1 / out return value
%r26	%i2	in parameter 2 / out return value
%r27	%i3	in parameter 3 / out return value
%r28	%i4	in parameter 4
%r29	%i5	in parameter 5
%r30	%i6 / %fp	frame pointer
%r31	%i7	return address - 8

2.1 Register Windows

As shown in **Table 1**, there are four groups of integer registers in SPARC architecture: *global* registers, *in* registers, *local* registers, and *out* registers. In a function call, the in registers have the incoming arguments from caller, local registers are for callee's local usage. Thus in and local registers are callee-save registers. The out registers are used to pass outgoing arguments to the function to be called. The contents of global registers do not change during the function calls.

The `save` and `restore` instructions are used to pass arguments and return values between caller and callee during a function call. When a `save` is issued, caller's in and local registers are saved, and the out registers are mapped to callee's in registers. When a `restore` is issued during the function exit, callee's in register are remapped back to caller's out registers.

However, the implementation of the `save`/`restore` mechanism uses a technique called *register window*. The register windows typically consists of 8 windows with 24 registers (in, local, and out registers) per window. Then in and out registers are actually slides (i. e., to be renamed) instead of copying during the function calls. Therefore, register windows acts as a cache of current registers set and operates in parallel with the stack frame. This technique is designed to reduce memory load/store instructions during the procedure calls, especially for large application programs. In SPARC V8, the register windows is managed by privileged instructions, and this limitation is relieved by SPARC V9. **Table 2** summarizes the changes of registers and instructions related to the implementation of register windows.

Table 2 Register Windows Related Changes from SPARC V8 to V9

		SPARC V9	SPARC V8
Registers			
Window invalid mask	-		WIM
Current window*	○	CWP	CWP
Processor state	△	PSTATE	PSR
Savable windows	+	CANSAVE	
Restorable windows	+	CANRESTORE	
Other windows	+	OTHERWIN	
Clean windows	+	CLEANWIN	
Window state	+	WSTATE	
Instructions			
Save and restore	△	SAVE/RESTORE	SAVE/RESTORE
State after RESTORE	+	RESTORED	
State after SAVE	+	SAVED	
Flush windows	+	FLUSHW	
Read and write WIM	-		RDWIM/WRWIM
Read and write PSR	-		RDPSR/WRPSR

○: unchanged, △: changed, +: added, -: deleted, *: privileged

As a result, the contents in register windows are different for each thread context and they must be flushed to each thread's own save area respectively. While user must use a trap `ta ST_FLUSH_WINDOWS` to flush the register windows in SPARC V8, SPARC V9 provides a more efficient non-privileged instruction `FLUSHW` to perform this operation [14]a.

2.2 Stack Layout

Figure 1 shows the stack layout of SPARC V9 convention. The SPARC V9 stack grows from high address to low address and must be 8-byte aligned. It also requires a bias of 2047 to be added to stack offset related to %sp, which is basically used as a flag bit (since 2047 is odd, the low bit of an address is 1) to distinguish from SPARC V8 stack. Different from other machine architectures, stack manipulation should no be operated on the top of stack, because a reserved area from the stack top is required by the kernel for its usage. For example, on SPARC V9, the top of %sp+2047+120 bytes is reserved area, which *must not be* clobbered by users. Users can save thread context to the address from %sp+2047+128.

The area from %sp+2047 to %sp+2047+120 is called kernel save window that is used to save callee's register window during the flushing of register windows. Both local and in registers are saved in this area, by which the frame pointer %fp or %i6 is chained via the kernel save area. Therefore, if a `restore` instruction cause an underflow trap, the frame pointer in the current register window becomes the stack pointer and corresponding contents in kernel save area can be reloaded. Accordingly, the reloaded registers contains a frame pointer to next frame of its caller and thus a series of historical registers sets are threaded down to the stack.

3. Context Switching for SPARC

3.1 Generic Context Switching

Generally, a context switching is to save the state of old thread and restore the state from new thread. David and Peter have described the stacks, registers and the implementation of user-space

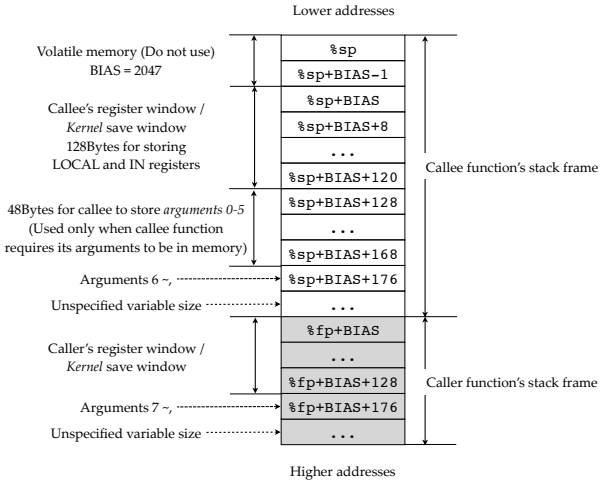

Fig. 1 Stack Layout of SPARC V9 Architectures

```
1  #define FRAMESIZE 176   /* must > 136    */
2  #define STACKBIAS 2047  /* SPARC V9 ABI */
3  #define SAVE_FP   128
4
5  typedef struct context
6  {
7    uint64_t sp;
8  } context_t;
9
10 void makecontext(context_t ctx,
11   void *stack, size_t stacksize)
12 {
13   uint64_t stack_tail = (uint64_t) stack;
14   uint64_t *fp;
15   stack_tail &= 0xFFFFFFFFFFFFFFF0;
16   ctx->sp = stack_tail - FRAMESIZE * 2
17     - STACKBIAS;
18   fp = (uint64_t *)
19     (ctx->sp + STACKBIAS + SAVE_FP);
20   *fp = (uint64_t)
21     (stack_tail - FRAMESIZE - STACKBIAS);
22 }
```

Fig. 2 C Code of makecontext() for Assembly Implementation

threads on the SPARC V8 architecture [7], [9]. However, their approaches cannot be straightforwardly applied to the SPARC V9 architecture because SPARC V9 has diffrent registers, instruction set, and stack layout than SPARC V8.

We implemented both context switching for both SPARC V8 and V9. In following sections, we illustrate our approaches by using SPARC V9 version as an example.

3.2 Synthesising Thread Stack

To perform context switching, the first step is to make a context for current thread by synthesising a thread stack. **Figure 2** shows the implementation of making an empty thread stack for SPARC V9, where `stack` and passed to `makecontext()` specifies a piece of continuous memory that grows from high address to low address with a given size of `stacksize`.

(1) The stack address is first aligned properly according to SPARC V9 requirements (line 15).
(2) Create two continuous stack frames from bottom high address and add up the stack bias required by SPARC V9 (line 16–17). The reason of using two frames is that, besides current frame, in the end of context switching functions (i. e., `setcontext()` or `swapcontext()`), a `restore` instruction is invoked to return back to an additional chained frame (see Section 3.3).
(3) Specify the address of saved %fp in callee's argument save area and set its content to the address of next chained frame (line 18–21, also see Fig. 1).

3.3 Saving and Restoring Registers

For synchronous (non-preemptive) context switching, only stack pointer (%sp), frame pointer (%fp), and return address (%i7) need to be saved. In most cases, floating-pointer (%f0~%f31) registers and global registers (%g0~%g7) are caller save registers. In, local, out registers (%i0~%i7, %l0~%i7, %o0~%o7) are cached in register windows and thus must be flushed to chained stack frames.

We use **Fig. 3**, the assembly implementation of `swapcontext()` for SPARC V9, to illustrate saving and restoring operations in context switching.

(1) By function calling convention [14], a `save` instruction rotates the register window to a new one and make a new frame on the stack at the same time (line 12).
(2) Save current/old %sp, %fp, and %i7 to the address specified by %i0, where %i0 corresponds to the `from` argument passed from caller (line 13–15). Note that the save offsets of %fp and %i7 are in callee's arguments saving area (see Fig. 1).
(3) Now it is safe to flush register windows to current stack by the SPARC V9 *non-privileged* instruction `FLUSHW` (line 16). For SPARC V8, this instruction should be replaced with `ta ST_FLUSH_WINDOWS`.
(4) Load new %sp, %fp, and %i7 from the address specified by %i1, where %i1 corresponds to the `to` argument passed from caller (line 17–19).
(5) Restore from the new stack and rotates the register window back to the caller's one by `restore` instruction (line 20). Note that after this point %i7 is remapped to %o7 in caller's context.
(6) Jump to the return address specified by %o7 and continue execution on new stack (line 21).

3.4 Using Library Context Switching Routines

Context switching can be also implemented by interfaces provided by C library, such as ucontext series (see man ucontext(3)). **Figure 4** shows the implementation of `makecontext()` and `swapcontext()` by using C library functions denoted as `libc_makecontext()` and `libc_swapcontext()`, which simply follows the usage convention of ucontext series.

3.4.1 Thread Local Storage

On SPARC architecture, the %g7 register is used as the pointer of thread local storage [3]. As a result, %g7 must not be clobbered during the context switch.

Unfortunately, the Linux kernel currently installed on FX10 (i. e., Linux 2.6.25) includes a bug of clobbering the thread local

```
#define FRAMESIZE 176
#define BIAS 2047
#define SP 128
#define I7 136

//swapcontext(context_t from, context_t to);

.p2align 4
.global swapcontext
.type swapcontext, @function
swapcontext:
  save %sp, -FRAMESIZE, %sp
  stx %sp, [%i0]
  stx %fp, [%sp+BIAS+SP]
  stx %i7, [%sp+BIAS+I7]
  flushw
  ldx [%i1], %sp
  ldx [%sp+BIAS+SP], %fp
  ldx [%sp+BIAS+I7], %i7
  restore
  jmpl %o7+8, %g0
.size swapcontext, .-swapcontext
```

Fig. 3 Assembly Implementation of swapcontext()

```
typedef struct context
{
  ucontext_t uc;
} context_t;

void makecontext(context_t ctx,
  void (func*) void,
  void *stack, size_t size)
{
  uintptr_t stack_start =
    ((uintptr_t) stack) -
    (size - sizeof(void*));
  libc_getcontext(&ctx->uc);
  ctx->uc.uc_stack.ss_sp = stack_start;
  ctx->uc.uc_stack.ss_size = size;
  ctx->uc.uc_link = NULL;
  int fn_ints[2];
  memset(fn_ints, 0, sizeof(fn_ints));
  memcpy(fn_ints, &func, sizeof(void*));
  libc_makecontext(&ctx->uc,
    (void(*)())voidcall_context_ep,
    2, fn_ints[0], fn_ints[1]);
}

void swapcontext(context_t from, context_t to)
{
  PRESERVE_TLSREG(to);
  libc_swapcontext(&from->uc, &to->uc);
}
```

Fig. 4 ucontext Implementation of makecontext() and swapcontext()

storage register [*1] , which has been fixed since Linux 2.6.26 [4]. Accordingly, we applied a user-level patch by reverting the effect of kernel saved %g7, as shown in **Fig. 5**. This preserving operation is then put *right before* each invoked context switch routine (i. e., `setcontext()` and `swapcontext()`) to take effect, as shown in line 27 in Fig. 4.

3.4.2 Using setjmp()/longjmp()

According to the source code of Glibc-2.7[*2], `setjmp()` and `longjmp()` are essentially implemented by `getcontext()` and `setcontext()`. Therefore, using `setjmp()`/`longjmp()` to implement context switch should expect equivalent performance as using ucontext functions.

[*1] Line 50 of linux-2.6.25/arch/sparc64/kernel/signal.c
[*2] Glibc-2.7/sysdeps/unix/sysv/linux/sparc/sparc64/[set,long]jmp.S

```
#ifdef __sparc_v8
#define PRESERVE_TLSREG(ctx) \
  asm volatile("st %%g7,[%0]" \
  ::"r"(&ctx->uc.uc_mcontext.gregs[REG_G7]) \
  : "memory")
#elif __sparc_v9
#define PRESERVE_TLSREG(ctx) \
  asm volatile("stx %%g7,[%0]" \
  ::"r"(&ctx->uc.uc_mcontext.mc_gregs[MC_G7]) \
  : "memory")
#endif
```

Fig. 5 User-Level Patch of TLS Bug for Linux 2.6.25

Table 3 Configuration of PRIMEHPC FX10 System

Compute Node: FUJITSU PRIMEHPC FX10	
CPU	SPARC64 IXfx 1.848GHz, 16 cores, L1 32KB, L2 12MB
Memory	SDRAM DDR3-1333ECC, 32GB/node, 85GB/sec
OS	XTCOS (Linux 2.6.25.8 based)
Build	GLIBC 2.7, GCC 4.2.4, GDB 6.6
Log-in Node: FUJITSU PRIMERGY RX300 S6	
CPU	Intel Xeon L5640 2.27GHz, 24 cores, 12MB cache
Memory	SDRAM DDR3-1333ECC, 48GB/node
OS	Red Hat Enterprise Linux 2.6.32
Build	GLIBC 2.12, GCC 4.4.5, GDB 7.2

4. Evaluation

4.1 Experimental Environments

Table 3 shows the hardware and software configurations of the FX10 system we used, up to date November 2012 [8]. Note that the versions of OS kernels and libraries do matter because the bugs and performance considerations we discuss in this paper are highly implementation-specific problems.

We conduct the experiments on *only one single* computing node or one log-in node, because the measure of context switching is sufficient within one node for shared memory. The evaluation of implementation and execution performance on multiple nodes will be included in our future work.

4.2 Context Switch Performance

Figure 6 presents the comparison of execution time of Fibonacci(30) on one single compute node, scaling from 1 to 16 threads. The implementation using ucontext functions (denoted as *uctx*) is over 30x slower than the implementation in assembly (denoted as *asm*). Furthermore, using `ta ST_FLUSH_WINDOWS` to flush register windows has over 4x overhead than using one single non-privileged `FLUSHW` instruction.

The source-level analysis shows that the overheads of using ucontext mainly come from following places,

- Using kernel trap `ta ST_FLUSH_WINDOWS` to flush register windows instead of a more efficient `FLUSHW` instruction.
- The library functions use kernel traps (trap 0x6e and 0x6f) instead of user-level code to manipulate thread context, which cross the kernel/user boundaries and introduce addi-

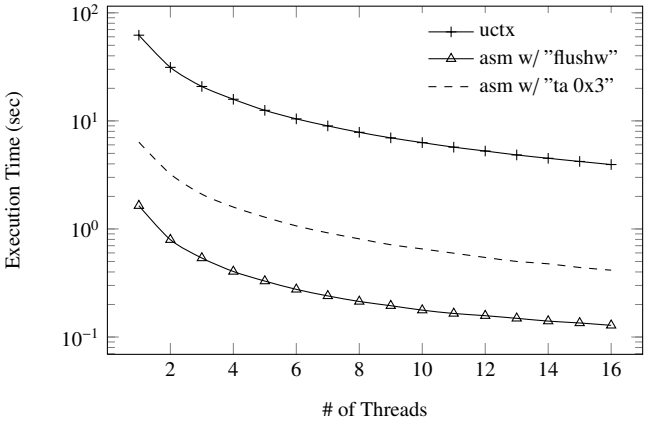

Fig. 6 Comparison of Execution Time of Fibonacci(30)

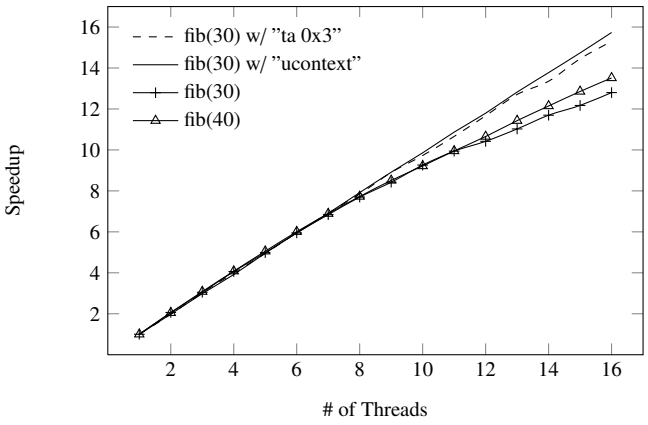

Fig. 7 Comparison of Scalability of Fibonacci

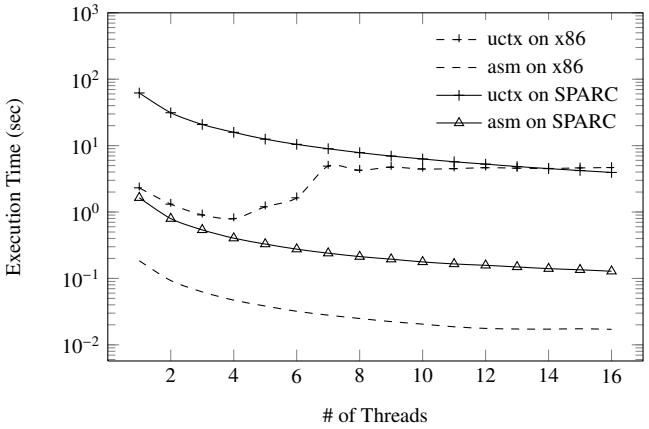

Fig. 8 Comparison of Fibonacci(30) on x86 and SPARC

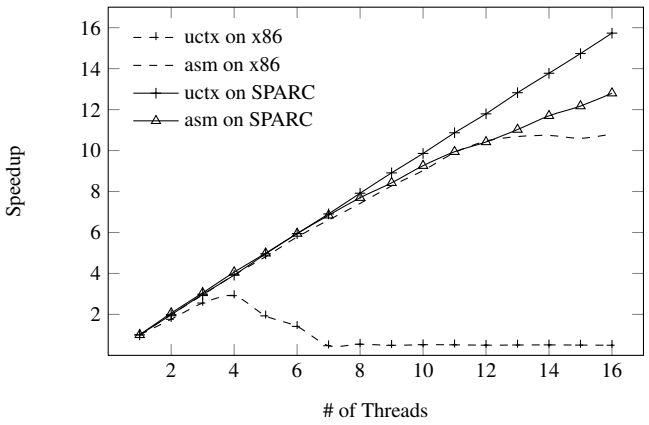

Fig. 9 Comparison of Scalability of Fibonacci(30) on x86 and SPARC

tional overhead. [*3].

- The kernel functions `sparc64_set_context()` and `sparc64_get_context()` always get and put (i. e., copying) registers contents from and to userland, which are heavy operations. [*4].
- The `sparc64_set_context()` function invokes a function `flush_user_windows()` that walks and saves each window respectively to flush register windows, instead of using one single assembly instruction to conduct register windows flushing job[*5].

As shown in **Fig. 7**, ucontext implementation and using `ta ST_FLUSH_WINDOWS` scales better than assembly implementation using `FLUSHW` instruction when the number of cores is large than 8. This is because they spend more CPU cycles on context switching. Figure 7 also shows that when the number of tasks increases such as in Fibonacci(40), the scalability improves.

To compare with the implementation for x86 architecture. We run the same benchmark on FX10 log-in node for up to 16 cores. **Figure 8** and **Figure 9** show that the ucontext implementation of x86 has a poor performance and only scales up to 5 threads. The assembly implementation of x86 has best performance but saturated when the number of threads reaches 12. Note that the CPU frequencies of computing node and login node are different (1.8GHz vs. 2.27GHz), we can conclude that the assembly implementation for SPARC is as efficient as the assembly implementation for x86.

To verify the analysis in Section 3.4.2, we use a simple thread library [6] that includes both a ucontext and a setjmp/longjmp implementation, to compare the performance of using different context switch routines with a Fibonacci-like micorobenchmark. **Figure 10** shows that the context switch overhead is almost the same for the SPARC implementation of ucontext and setjmp/-longjmp. However, on x86, the implementation of ucontext and setjmp/longjmp is different and thus their execution time varies significantly.

4.3 Performance as Chapel Tasking Layer

The Chapel language is an object-oriented parallel programming language that is designed to improve programmability for modern HPC systems with significant parallelism [2], [13]. The Chapel project was started by Cray as part of DARPA's HPCS programme. Chapel adopts the *global view model* rather than conventional the fragmented model (which is derived from the SIMD model). It supports the abstractions of task parallelism, data parallelism, and nested parallelism. The parallelism is described using an *implicit multithreading* scheme, in which independent computations are mapped to a collection of threads.

[*3] Glibc-2.7/sysdeps/unix/sysv/linux/sparc/sparc64/[get, set]context.S
[*4] Linux-2.6.25/arch/sparc64/kernel/signal.c
[*5] Line 14 of linux-2.6.25/arch/sparc64/kernel/signal.c

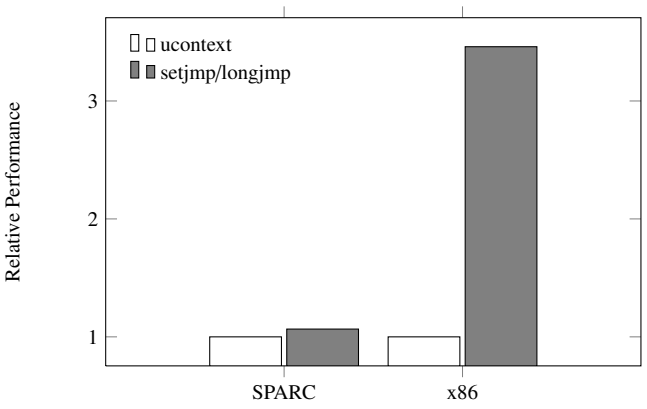

Fig. 10 Comparison of Performance of Library ucontext and setjmp

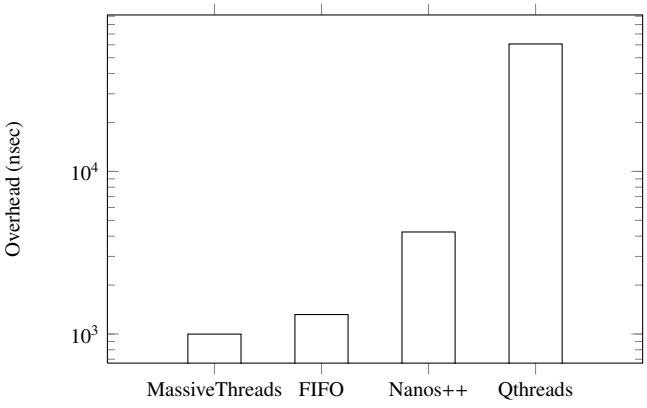

Fig. 11 Comparison of Overhead to Create and Join a Task

Low-level thread management (e. g., create and join) is implemented by the compiler and runtime system instead of users. The unit of physical computation resources, i. e., CPU plus memory, is abstracted as the *locale* type in Chapel.

Chapel integrated several thread libraries as its low-level tasking layer, including Pthreads, Nanos++ [5], Qthreads [15], and MassiveThreads. To compare with these tasking layers, we use several benchmarks to measure their performance. In following experiments, we uses Chapel 1.6.0 release but replaces the default MassiveThreads with our current version with SPARC assembly support. The configuration of Chapel is set to use one single locale without communication channel, i. e., `CHPL_COMM=none`.

Figure 11 shows the overhead to create one task and wait for termination. In this experiment, the number of actual worker thread is set to one but assigned with many tasks, which allows us to exclude the overhead from thread scheduling when multiple worker threads are enabled. MassiveThreads is the fastest out of four tasking layers.

To further investigate the overhead of thread library itself, we measure the same overhead but using a C benchmark program linked with native thread libraries. **Figure 12** shows the results.

We used three recursive task-parallel applications to conduct the comparison: Fibonacci(30) calculation, quicksort of 2M elements array, and cache-oblivious matrix multiplication for 1000× 1000 matrix. **Figure 13**, **Figure 14**, and **Figure 15** show the results, respectively. In Fig. 13 and Fig. 14, the performance results

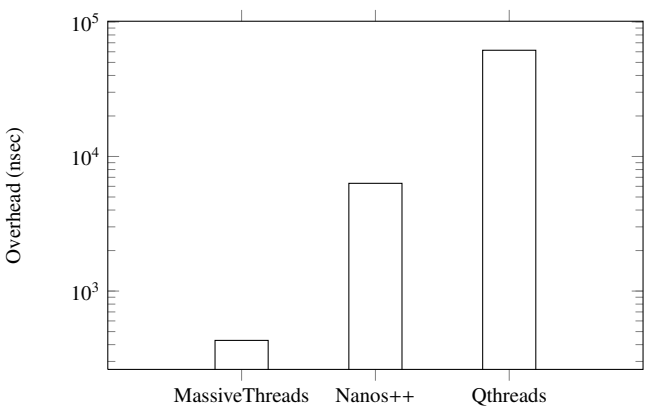

Fig. 12 Overhead to Create and Join a Task of Thread Library

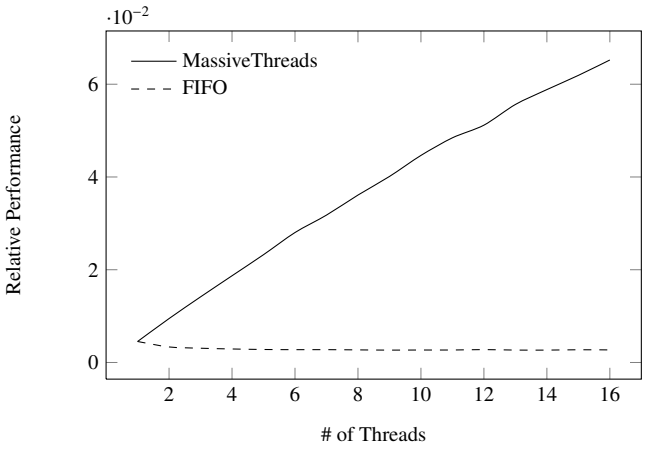

Fig. 13 Comparison of Fibonacci(30)

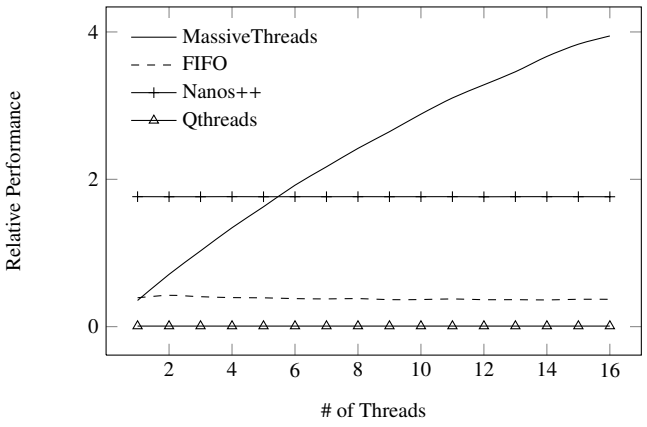

Fig. 14 Comparison of Quicksort(2M)

are relative to the performance with tasking layer none, i. e., tasking layer by sequential function calls. Results omitted are experiments that took too long time to finish.

For all benchmarks, MassiveThreads shows best performance and scalability among these four tasking layers. For other tasking layer, they do not include an assembly implementation of context switching but use library functions, which can be one important cause for their significant overhead.

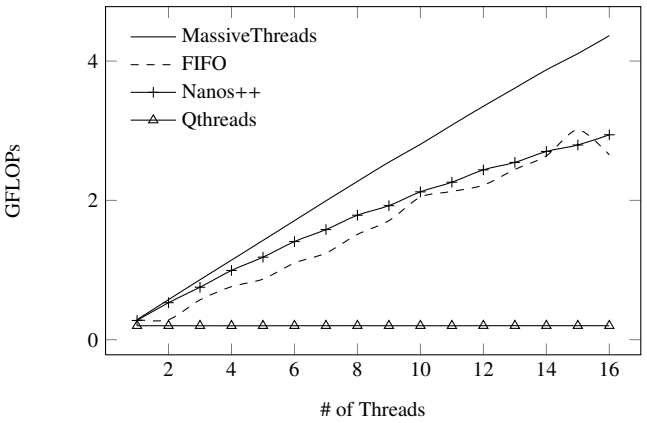

Fig. 15 Comparison of Matrix Multiplication ($10^3 \times 10^3$)

5. Conclusions and Future Work

We ported MassiveThreads thread library to FX10 supercomputer system by implementing an efficient context switching in assembly. On the SPARC architecture, register windows management has significant performance impact on context switching. Implementation should take advantage of the non-privilege instruction `FLUSHW` provided by SPARC V9 to boost the performance. Programmers should also be aware of the kernel save area in the stack and thread local register %g7 to keep context switch safe.

Current Linux kernel implementation (i. e., Kernel 2.6.34 and 3.6.2) of ucontext functions for SPARC V9 is out-of-date . This implementation includes significant overheads of context switch. Other thread libraries and user applications, whatever use ucontext to perform context switch, will inherit these overheads accordingly.

In the future, our goal is to enable MassiveThreads to run on multiple nodes (i. e., in distributed memory settings) of FX10 with the support from other middleware or just as the tasking layer of Chapel language.

Finally, the source code of our implementation is public online available as a part of MassiveThreads repository at http://code.google.com/p/massivethreads/.

Acknowledgments This work is partially supported by JST, CREST through its research project: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

[1] Ajima, Y., Sumimoto, S. and Shimizu, T.: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers, *Computer*, Vol. 42, pp. 36–40 (2009).
[2] Chamberlain, B. L., Callahan, D. and Zima, H. P.: Parallel Programmability and the Chapel Language, *International Journal of High Performance Computing Applications*, Vol. 21, pp. 291–312 (2007).
[3] Drepper, U.: ELF Handling for Thread-Local Storage, Red Hat Inc. (online), available from ⟨http://people.freebsd.org/ alfred/tls.pdf⟩ (accessed 2012-11-14).
[4] Filardo, N.: SPARC64 get/setcontext smashes TLS, Todo (online), available from ⟨http://sourceware.org/bugzilla/show_bug.cgi?id=6577⟩ (accessed 2012-11-14).
[5] Group, B. P. M.: Nanos++ Runtime Library, Barcelona Supercomputing Center (online), available from ⟨https://pm.bsc.es/projects/nanox⟩ (accessed 2012-11-14).
[6] Jones, E.: Implementing a Thread Library on Linux, Todo (online), available from ⟨http://www.evanjones.ca/software/threading.html⟩ (accessed 2012-11-14).
[7] Keppel, D.: Register Window and User-Space Threads on the SPARC, Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington (1991).
[8] Limited, F.: FX10 Supercomputer System, The University of Tokyo (online), available from ⟨http://www.cc.u-tokyo.ac.jp/system/fx10/⟩ (accessed 2012-11-14).
[9] Magnusson, P. S.: Understanding Stacks and Registers in the SPARC Architecture, Swedish Institute of Computer Science (online), available from ⟨http://www.sics.se/ psm/sparcstack.html⟩ (accessed 2012-11-14).
[10] Nakashima, J., Dun, N. and Taura, K.: MassiveThreads Thread Library, The University of Tokyo (online), available from ⟨http://massivethreads.googlecode.com/⟩ (accessed 2012-11-14).
[11] Nakashima, J. and Taura, K.: Multithread Framework that Manages both Efficient I/O and Lightweight Thread Management, *Information Processing Society of Japan Transactions on Programming*, Vol. 4, No. 1, pp. 13–26 (2011).
[12] SPARC Internationl, I.: The SPARC Architecture Manual Version 8, SPARC Internationl, Inc. (online), available from ⟨http://www.sparc.com/standards/V8.pdf⟩ (accessed 2012-11-14).
[13] Team, C. D.: Chapel Programming Language, Cray Inc. (online), available from ⟨http://chapel.cray.com/⟩ (accessed 2012-11-14).
[14] Weaver, D. L. and Germond, T.: The SPARC Architecture Manual Version 9, SPARC Internationl, Inc. (online), available from ⟨http://sparc.org/standards/SPARCV9.pdf⟩ (accessed 2012-11-14).
[15] Wheeler, K., Murphy, R. and Thain, D.: The Qthread Library, Sandia National Laboratories (online), available from ⟨http://www.cs.sandia.gov/qthreads/⟩ (accessed 2012-11-14).