

# MassiveThreads: A Thread Library for High Productivity Languages

Jun Nakashima and Kenjiro Taura

The University of Tokyo  
Tokyo, Japan  
`nakashima@eidos.ic.i.u-tokyo.ac.jp`  
`tau@eidos.ic.i.u-tokyo.ac.jp`

**Abstract.** An efficient implementation of task parallelism is important for high productivity languages. Specifically, it requires a tasking layer that fulfills following requirements: (i) its performance scales to high core counts, and (ii) it is seamlessly integrated into a runtime system that performs inter-node communication and synchronization. More specifically, it should facilitate interactions between tasks and threads dedicated for inter-node communication. There have been many implementations that satisfy (i), but, to the best of our knowledge, none of such systems satisfy both requirements.

To address this issue, we propose a thread library called MassiveThreads. It provides not only lightweight threads and a scalable dynamic load-balancing mechanism among CPU cores, but also Pthread-compatible API and I/O semantics. In MassiveThreads, issuing a blocking I/O call triggers a user-level context switch instead of blocking the underlying OS-level thread. These features simplify interactions between tasks and communication threads by instantiating both of them on top of MassiveThreads.

## 1 Introduction

Current parallel programming languages and frameworks such as MPI[7] provide programming models based on the primitive abstraction of hardware. They achieve high performance by putting the burden of managing tasks and communication on programmers. The burden is becoming heavier as machines become larger, more heterogeneous, and more hierarchical.

To address this issue, there have been recently proposed many parallel languages that aim to improve both performance and productivity. Many of them provide a global address space and general *task parallelism*, in which tasks can be nested and created at arbitrary points of execution. This general form of task parallelism encompasses many syntactically different forms of parallelism supported in parallel programming languages, including fork-join, parallel recursions and parallel for loops.

The implementation of an application with task parallelism becomes simple and modular if there is an underlying thread package that can directly map

multi-tasking primitives of the language to multi-threading primitives of the thread package. For example, if the underlying thread package is Pthreads, a task creation is directly translated into `pthread_create`. But instantiating a Pthread for each task creation performs poorly especially when a large number of tasks are created. To perform well when executing many fine-grained tasks, the underlying thread package should be lightweight.

A basic technique for implementing efficient task parallelism is known. Rooted back to Lazy Task Creation [8] proposed for a parallel Lisp, many systems are based on a similar principle of work-first and LIFO scheduling within each worker (underlying OS-level thread to execute tasks) and FIFO task stealing between workers. However, most of them assume applications running on single node machines.

Many high productivity languages such as X10[4] and Chapel[3] support task parallelism and work on distributed memory machines. If their runtime system can switch tasks triggered by communication, overlapping communication and computation — one of the important techniques to achieve good performance on distributed memory machines — can be easily written with the languages by simply creating many tasks.

One way to implement such runtime system is to integrate existing lightweight thread packages into a runtime system that communicates with other nodes, but it is complicated. Suppose a lightweight thread implemented as a user-level thread performs a blocking I/O call. In this case, not the lightweight thread but the underlying OS-level thread is blocked until the I/O call finishes thus some degree of parallelism is lost. This issue can be addressed by yielding all the communication to a dedicated OS-level thread, but in this case synchronization between user-level tasks and the OS-level communication thread becomes a non-trivial issue.

Our approach is to eliminate the problem at its root: a thread package that is lightweight and can handle blocking I/O calls without blocking the underlying OS-level threads. By executing both a communication thread and lightweight threads for task parallelism on top of this thread package, integration into the runtime system can be simplified. Our proposed solution is implemented as a library called MassiveThreads, which we describe in this paper. This paper also describes the evaluation result of its performance.

## 2 Related Work

Comparing to OS-level threads, user-level threads have two major advantages. (1) User-level threads' overheads are much smaller since thread management does not require system calls, and (2) User-level threads can use scheduling policies optimized for specific applications. Such an application that can take advantage of user-level threads is a task parallelism runtime system. As described in previous section, efficient implementations of user-level threads for task parallelism on a shared memory machine is a well studied topic. There are many languages, frameworks, and libraries that support task parallelism by lightweight

threads, such as Cilk[1], Java Fork/Join Framework[5], Intel Threading Building Blocks (TBB)[10], StackThreads/MP[12], Qthreads[14], and Nanos++[2].

Another application of user-level thread is for processing concurrent I/O. Mapping a thread for each connection is a naive way for implementing concurrent I/O processing, but OS-level thread is too heavyweight for this purpose. Capriccio[13] and StateThreads[11] provide user-level threads that can automatically switch the context triggered by I/O calls to leverage highly concurrent server implementations.

### 3 Design and Implementation

#### 3.1 Design Overview

To make threads lightweight, MassiveThreads is implemented as a user-level thread library. In order to handle I/O calls without blocking the underlying OS-level threads, it automatically intercepts blocking I/O calls and switches contexts to other ready threads.

The MassiveThreads library is build as a shared library that provides the functions compatible with Pthreads. Therefore it can be used in place of Pthreads by simply linking it instead of Pthreads, or by dynamically loading at runtime by using environment variables. Thanks to this feature, existing communication libraries for Pthreads can easily run on top of MassiveThreads.

#### 3.2 Definition of Terms

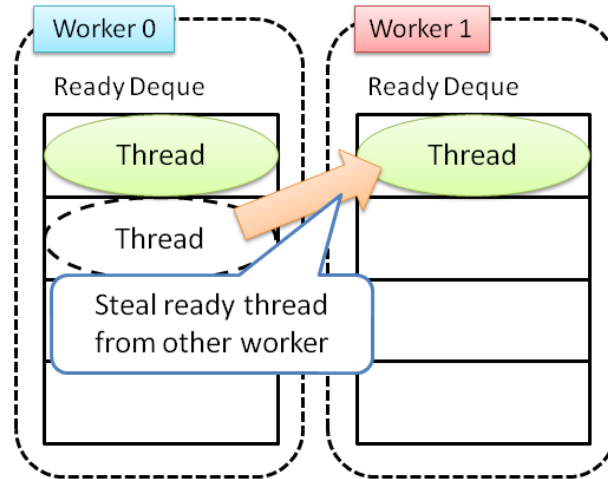
- “thread” means user-level thread managed by the MassiveThreads library
- “worker” means OS-level thread to execute user-level threads
- “deque” means double-ended queue

#### 3.3 Thread Scheduling

**Design** We chose work-first and LIFO scheduling within each worker and FIFO randomized work stealing between workers as MassiveThreads scheduling policy for two reasons. First, this scheduling policy is known to be efficient for recursive task parallelism. Most parallel constructs including fork-join and parallel for loop can be easily translated to recursive task parallelism. Therefore this scheduling policy can give the potential to execute most parallel constructs efficiently under the appropriately implemented compiler and the runtime system. The second reason is that the algorithm has no centralized components which may become a bottleneck with large number of cores.

**Data Structure** The MassiveThreads library creates workers and binds them to CPU cores. Each worker thread has a deque called ready deque to store the ready user-level threads (Fig.1). Ready deque supports the following 3 operations.

- push: Insert a thread to the head by owner
- pop: Get a thread from the head and delete it from the deque by the owner worker
- take: Get a thread from the tail and delete it from the deque by non-owner worker



**Fig. 1.** Data Structure to Execute User-level Threads

**Scheduling** When a new thread is created, underlying worker suspends a thread currently running, inserts it to the head of the ready deque by push operation, and executes the new thread immediately. When the current running thread is finished, the worker obtains a new thread using pop operation. If there is no thread in the ready deque, the worker tries to steal a ready thread from randomly chosen workers' ready deque using take operation.

**Optimization** The MassiveThreads library is implemented to minimize context switching and thread creation overhead. This section describes some of these optimizations and their corresponding performance effects are shown in Section 4.

*Avoid Using ucontext* On most Linux systems, portable user-level context switching library called *ucontext* can be used to implement user-level threads. We first use it for its portability, but found that its large switching overhead due to internal system calls for switching signal masks became a serious bottleneck.

To address this issue, we implemented context switching routines that switch callee-saved registers only.

*Ready Deque Implementation* To implement a ready deque, we followed similar approaches to Cilk[1] and Java Fork/Join Framework[5]. A ready deque consists of an array to store threads, two integers pointing the head and tail of the deque, and one spinlock. Following is the brief description of how it works.

- Push operation first stores a new thread to the array, and then increments the head pointer.
- Pop operation decrements the head pointer, and compares the head and tail pointers to check whether it is safe to return the result without locking. If it is safe result is returned without locking, otherwise lock is acquired to avoid conflicts.
- Take operation acquires the lock to serialize other operations, increments the tail pointer, and then checks conflicts with pop operation. If there is no conflict the result is returned.

To work the algorithm correctly, memory accesses order must not be changed. To meet this requirement we inserted memory barrier instructions and code snippets to suppress memory access re-ordering by the compiler.

With this algorithm push operation can be done without locking, and if there are more than one thread in the ready deque, pop operation can also be done without locking.

Additionally, we applied double-checked locking optimization to pop and take operations. Before pop and take, the number of threads in the ready deque is checked by comparing the head and tail pointer. If it is zero operations are aborted because there seems to be no thread available. This optimization improves the load balancing ability because it increases the amount of work stealing attempts per unit time by reducing the overhead of failed work stealing attempts.

*Double-Checked Locking on Joining a Thread* Join function waits for the termination of a thread and returns the exit value of the thread. When a thread is terminated, its exit value is stored to the thread descriptor — an internal data structure to describe the thread. At the time the thread is joined by the other thread, the exit value is read and the descriptor is released. Thread termination and join function call can be occurred at the same time, thus lock is required to avoid race conditions.

In order to support join function, a thread descriptor includes:

- Area to store the exit value
- An Integer to describe thread status (e.g. running, suspended, finished)
- A reference to the thread waiting for the termination
- A spinlock to avoid a race condition.

Before we applied double-checked locking optimization, join function is implemented as the following:

1. Acquire the lock of a target thread descriptor
2. Read the status
3. If it is already finished, read the exit value, release the descriptor, and return
4. Otherwise, set the reference to the currently running thread, suspend it, and then release the lock.
5. Read return value, release the descriptor, and return after the thread is continued

And thread termination is implemented as the following:

1. Acquire the lock of a target thread descriptor
2. Set the exit value
3. Set the status as finished
4. If the reference of suspended thread is set, resume it, and release the lock
5. Otherwise, get a thread from a ready deque and continue it, and release the lock.

This implementation acquires the lock 2 times for each thread join and termination. To reduce the number of attempts to acquire the lock, We applied double-checked locking optimization. Specifically, for thread termination, just after releasing the lock the status is set to the another one called “ready to be released”. For join function, before acquiring the lock the status is checked. If the status is ready to be released, it reads the exit values and releases the descriptor without acquiring the lock. Before releasing the descriptor, it waits for status to change to ready to be released. Appendix A shows pseudocode of join operation and thread termination with double-checked locking optimization.

This optimization is effective for most task-parallel application because when a thread joins the other thread, usually it has already terminated.

*Faster Thread-Local Storage Model* The MassiveThreads library internally uses thread-local storage to obtain worker-local information. There are 4 thread-local storage model: General Dynamic, Local Dynamic, Initial Exec, and Local Exec[6]. By default, General Dynamic model — the most general model in 4 models — is chosen. In General Dynamic model, access to the thread-local storage is compiled to a function call. We notice that this overhead takes up high percentage in thread creation and join overheads, because the thread-local storage is accessed very frequently. In order to reduce the overhead, we choose Initial Exec model — more restricted one than General Dynamic — through GCC compiler flags. In this model, access to thread-local storage is compiled to only a few instructions. This optimization has a drawback that the MassiveThreads library must be initially loaded. Thus runtime dynamic library loading mechanisms (e.g. dlopen) may not be used. But we believe this drawback is not serious, because most parallel applications do not load libraries dynamically, but libraries are linked with applications so that they are initially loaded.

### 3.4 Blocking I/O Call Handling

Currently this function only supports blocking I/O on network socket, thus in this paragraph, we use the socket-specific terms for explanation. But this implementation is essentially independent from the underlying I/O mechanism and can be applied to other blocking I/O calls. This implementation looks similar to that of Capriccio[13] or StateThreads[11], but is different in that it supports working on multiple workers.

**Data Structure** Fig.2 illustrates the data structures for I/O handling. To manage the threads waiting for a file descriptor to become ready, The MassiveThreads library assigns two lists called “blocked lists” for each file descriptor. One of the blocked list is for read and the other is for write.<sup>1</sup> A blocked list stores threads waiting for the file descriptor to become ready for the requested operation, as well as the arguments to the requested I/O call.

To check the status of file descriptors, the MassiveThreads library uses *epoll*. *epoll* is an I/O notification mechanism in Linux. To use *epoll*, file descriptors to check should be registered to an *epoll* instance. After that the list of ready file descriptors can be obtained using *epoll.wait* function. In order to distribute I/O handling operations, each worker has its own *epoll* instance. File descriptors are registered to one of the *epoll* instances.

To look up the blocked lists from a file descriptor, there is a map from a file descriptor to the corresponding blocked lists. It is implemented by a hash table and lookups to the map with different file descriptors can be done concurrently unless a hash collision occurs.

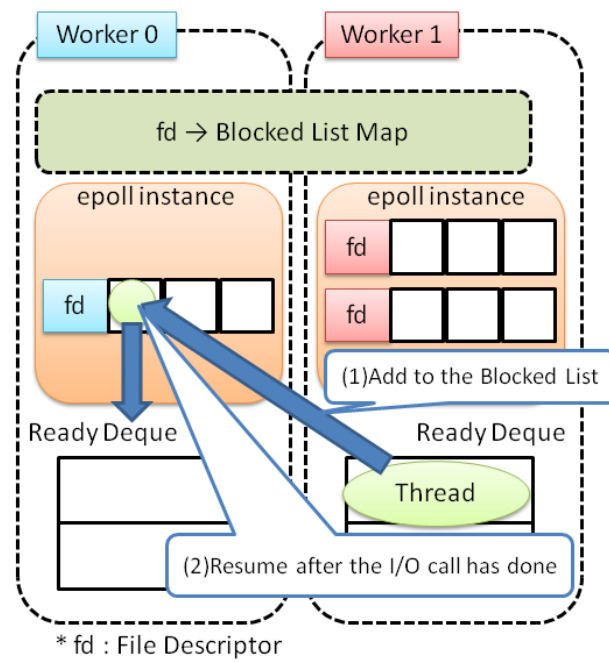
**I/O Handling Procedures** I/O handling in MassiveThreads consists of three procedures, namely, registering a new file descriptor, performing the I/O call, and polling to resume blocked threads.

*Registering a New File Descriptor* When a new file descriptor is created, two blocked lists are created and an association from the file descriptor to the lists is added to the map. Then it is registered to the *epoll* instance of the worker. To distribute I/O management loads, The worker is randomly chosen.

*Performing the I/O Call* When the application performs a blocking I/O call, it is intercepted by MassiveThreads, and performed with non-blocking option. If I/O call fails because the file descriptor is not ready, then the worker puts the caller thread and the arguments of the I/O call to the corresponding blocked list (Fig.2 (1)), and then switches to the thread in the ready deque.

---

<sup>1</sup> In Fig.2, only one blocked list is shown for each file descriptor for simplicity.



**Fig. 2.** Data Structure for I/O Handling



*Polling to Resume Blocked Threads* When a worker has no thread to execute, it checks the status of the file descriptors using *epoll*. If there is a ready file descriptor with a non-empty blocked list, then the worker tries the I/O call again. If it now succeeds, the thread in the entry is put into the head of the ready deque of the worker (Fig.2 (2)). We choose this policy in order to minimize migrating threads between workers by running a thread which use a file descriptor on a worker that checks its status as possible.

## 4 Evaluation

First, in order to confirm the MassiveThreads library has enough performance for leveraging task parallelism implementation, we evaluated the overheads to create and join one thread and load balancing ability. We also compared the performance with Cilk using practical applications. Then, we evaluated how well blocking system call handling works through ping-pong benchmark with many concurrent connections.

### 4.1 Thread Create and Join Overheads

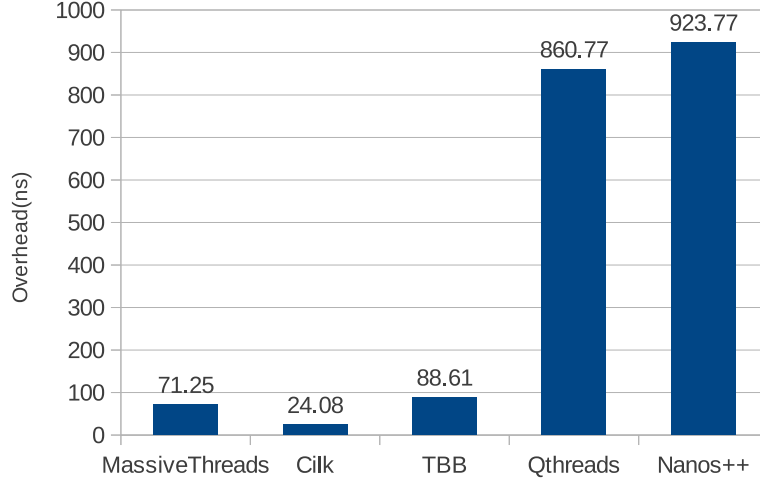
The overhead to create and join threads are especially important for fine-grained task parallelism. To evaluate them, we repeatedly create and join an empty (immediately finishing) thread using a single worker, and measured the overhead. The experimental setup is shown in Table 1.

**Table 1.** Experimental Setup for Overhead and Scalability Evaluation

CPU	Xeon E7540 (2.0GHz) 4 Sockets
OS	Debian Linux 2.6.32
Compiler	GCC 4.6.0
Cilk	version 5.4.6
Intel TBB	version 3.0
Nanos++	version nanox-e3a0ce4 (included in Chapel 1.4) NX_SCHEDULE=cilk
Qthreads	version 1.7 sherwood scheduler QTHREAD_NUM_WORKERS_PER_SHEPHERD=1

Fig.3 shows the overhead to create and join one thread. For comparison, this figure also shows that of Cilk, Intel Threading Building Blocks (TBB), Qthreads, and Nanos++. The overhead of MassiveThreads is about 70 nanoseconds, which is close to that of TBB.

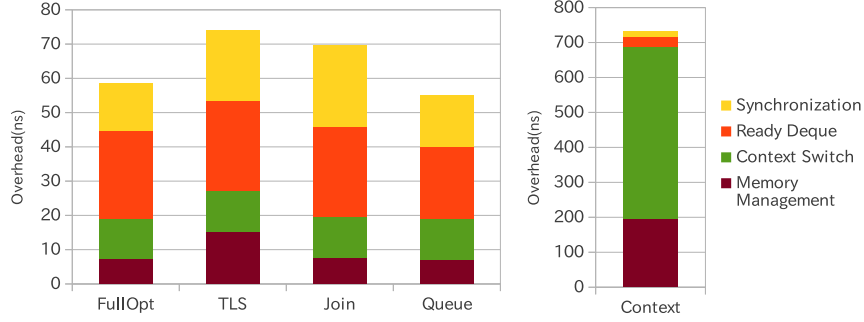
To see the overhead in more detail and to confirm the benefits of the optimizations described in Section 3.3 we broke down the overheads into 4 major parts:



**Fig. 3.** Overheads to Create and Join One Thread

memory management (allocation, initialization, and release for thread descriptor and stack), context switching, operation to the ready deque, and synchronization on thread join. Fig.4 shows the breakdown in the stacked bar graphs. *Fullopt* in the figure shows the overhead with all the optimization enabled. The other four bars show the overhead with one optimization disabled: choosing faster thread-local storage model on *TLS*, double-checking for thread join on *Join*, double-checking for ready deque operation on *Queue*, and avoid using *ucontext* for *Context*.

Without using faster thread local storage (see *Fullopt* and *TLS*), memory management and synchronization overhead increase, because most functions read worker-local data stored in the thread-local storage. When disabling double-checking for thread join (*Fullopt* vs. *Join*), synchronization overhead gets larger due to the extra atomic operation, while it is usually avoided when this optimization is enabled. Interestingly, ready deque operation overhead gets smaller when disabling double-checking for the ready deque operation (*Fullopt* vs. *Queue*). We are now investigating this case in order to reveal the cause and look for the opportunity to further performance improvement. When using *ucontext* instead of hand-written context switching functions (*Fullopt* vs. *Context*), the overhead jumps up to about 700 nanoseconds because of large increment of memory management and context switching overheads, because *ucontext* issues system calls every time on switching the contexts and on initializing the contexts for the new thread.



**Fig. 4.** Overhead Breakdown and Optimization Effects

## 4.2 Load Balancing on Unbalanced Tree Search

**Table 2.** UTS Benchmark Dataset

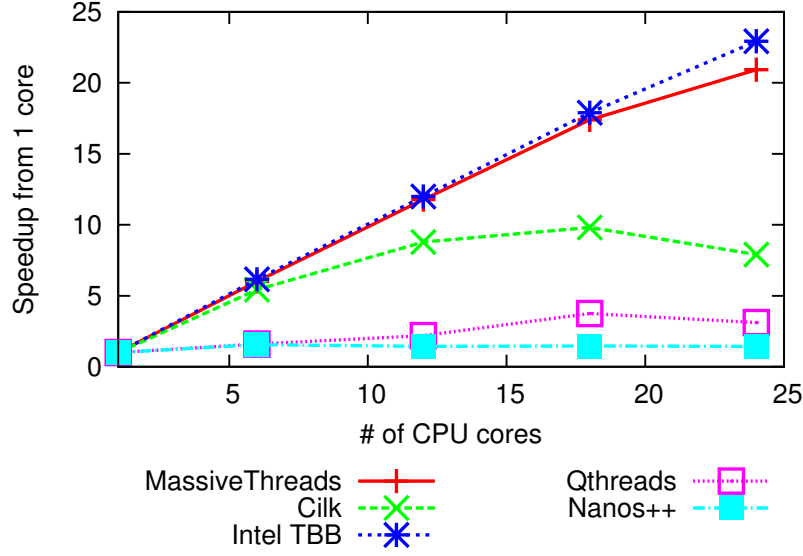
Tree	Type	$b_0$	$q$	$m$	$r$	Depth	Nodes
T3L	binomial	2000	0.200014	5	7	17844	$1.1 \times 10^8$

To evaluate the scalability, we use Unbalanced Tree Search (UTS) Benchmark[9]. This benchmark measures the performance of searching highly unbalanced tree. The tree shape is highly unbalanced but contains sufficient amount of parallelism enough to fully utilize many CPU cores. Therefore, this performance reflects the performance of dynamic load balancing. We parallelize the reference implementation that performs depth-first search (*uts-dfs*) by creating threads recursively. As a dataset, we choose *T3L* tree (details are shown in Table 2).

Fig.5 shows the speedup of MassiveThreads, Cilk, and Intel TBB, Qthreads, and Nanos++ relative to a single core performance of each implementation. MassiveThreads speedup factor is approximately 21. Except for using 24 cores, the performance is close to that of TBB, which performs the best in existing frameworks used for the evaluation.

## 4.3 Performance of Practical Programs

To see MassiveThreads performance for more practical applications, first we picked up programs from Cilk distribution that are non-interactive and can be directly translated to MassiveThreads (specifically, “abort” statement — task cancellation in Cilk — cannot be directly translated). Then we ported them to



**Fig. 5.** Speedup of UTS Benchmark

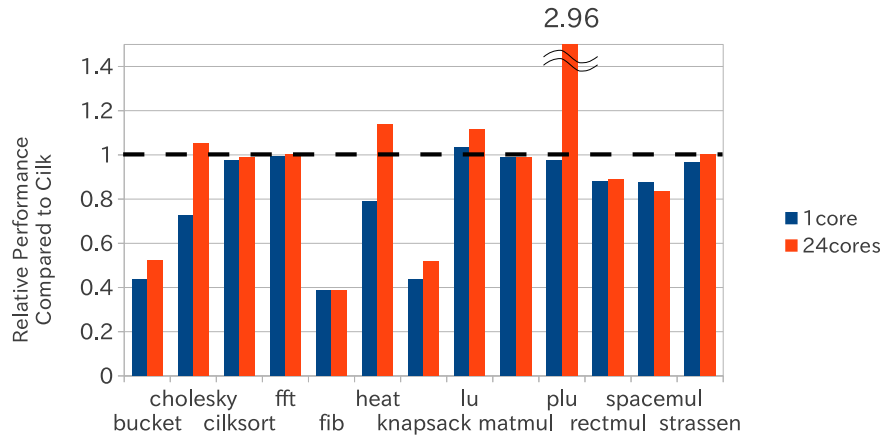
MassiveThreads and evaluated the performance. Programs and their arguments are shown in Table 3. Fig.6 shows the relative performance compared to Cilk on a single core and 24 cores. In most program, MassiveThreads performances are similar to or little worse than Cilk. But in *bucket*, *fib* and *knapsack*, the MassiveThreads library has much worse performance than Cilk. We guess the reason is that the task granularity is too small to hide MassiveThreads overhead which is about 3 times larger than Cilk. On the other hand, in *cholesky*, *heat*, *lu* and *plu*, The MassiveThreads library outperforms Cilk on high number of cores, because it scales well enough to compensate its larger overhead.

#### 4.4 Blocking I/O performance

To evaluate the blocking I/O performance, we use ping-pong benchmark between 2 nodes. One node runs a server and the other runs a client. In this benchmark, a server accepts TCP connections, and creates a thread for each connection. Threads in the server wait for a 1-byte message from the its connection. When a message arrives, the thread sends an acknowledgement. A client establishes connections to the server. In order to limit message concurrency, connections are distributed among the predefined number of threads. Threads in the client randomly choose a connection, send a 1-byte message, and wait for the reply. In this experiment, we limit the concurrency to 128. We define a pair of the message and the reply as a transaction, and measured the throughput of transaction. The experimental setup is shown in Table 4. Fig.7 shows

**Table 3.** Benchmark Parameters on Practical Programs

Name	Commandline Arguments	Description
bucket	-n 10000000	Bucket sorting
cholesky	-n 6000 -z 40000	Cholesky decomposition of sparse matrix
cilksort	-n 400000000	Sorting
fft	-n 268435456	FFT
fib	44	Fibonacci
heat	-g 1 -nx 6000 -ny 6000 -nt 400	Heat diffusion solver using jacobi iteration
knapsack	-benchmark long	0-1 knapsack solver using branch-and-bound
lu	-n 8192	LU decomposition of dense matrix
matmul	6000	Cache-oblivious matrix multiply
plu	-n 8192	LU decomposition with partial pivoting
rectmul	-x 8192 -y 8192 -z 8192	Rectangular matrix multiply
spacemul	-n 8192	Dag-consistent matrix multiply
strassen	-n 8192	Strassen's algorithm

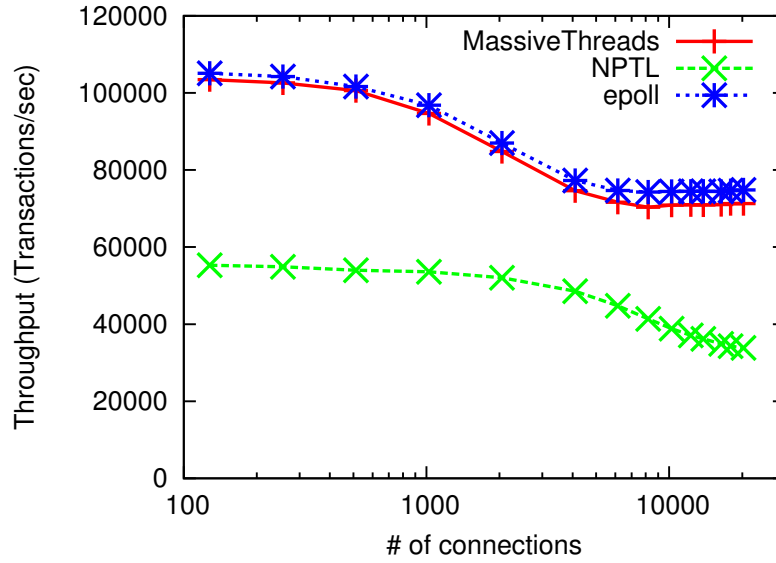


**Fig. 6.** MassiveThreads Relative Performance on Practical Programs

the throughput of MassiveThreads, NPTL (OS-level Pthreads on Linux), and single-threaded event-driven implementation using *epoll*. User-code for evaluation is common for both libraries. The MassiveThreads library outperforms NPTL and achieves close performance as *epoll*, which indicates MassiveThreads blocking system call handling works well enough to utilize *epoll* performance. Fig.8 shows the throughput using 8 cores. The MassiveThreads library achieves up to 4.5x better throughput than NPTL.

**Table 4.** Experimental Setup for Ping-pong

CPU	Xeon E5410 (2.5GHz) 4 Sockets
OS	Linux 2.6.26 (Debian)
Network	10Gbit Ethernet
C Compiler	GCC 4.4.1



**Fig. 7.** I/O Throughput of Ping-pong Benchmark using 1 core

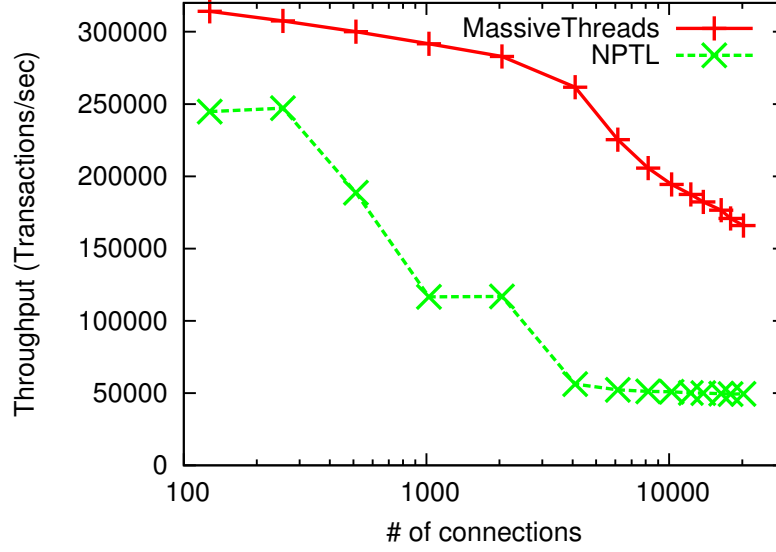


Fig. 8. I/O Throughput of Ping-pong Benchmark using 8 cores

## 5 Conclusion and Future Work

In addition to good performance on single node, task parallel runtime system for distributed memory environment is required to interact with inter-node communication libraries. Our approach to address this issue is to implement a thread library called MassiveThreads, which can execute both tasks and communication threads on top of it. To achieve this, it supports context switches triggered by blocking system calls and has compatible API and semantics with Pthreads.

Evaluation results show the MassiveThreads library has competitive performance with existing task parallel implementations, and ping-pong benchmark result show user-level context switches triggered by system calls can multiplex blocking socket I/O calls from many threads better than OS-level Pthreads.

As future work we are going to the following:

1. Perform more in-depth performance analysis.
2. Support context switching for more types of blocking system calls or I/O calls for interconnects, and evaluate the benefits.
3. Evaluate MassiveThreads on distributed memory applications and study scheduling policies for good interactions between tasks and communication threads.

MassiveThreads source code is available under BSD license from this URL:

– <http://googlecode.com/p/massivethreads>

## Acknowledgements

This work was supported by JSPS Grant-in-Aid for JSPS Fellows Grant Number 248391.

## References

1. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou: Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* **30**(8) (1995) 207–216
2. BSC: Nanos++ <http://pm.bsc.es/projects/nanox>
3. D. Callahan, B. L. Chamberlain, H. P. Zima : The Cascade High Productivity Language. In: in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04). (2004) 52–60
4. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, : X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 519–538
5. D. Lea: A Java Fork/Join Framework. In: JAVA '00: Proceedings of the ACM 2000 conference on Java Grande, New York, NY, USA, ACM (2000) 36–43
6. U. Drepper: ELF Handling for Thread-Local Storage.
7. Message Passing Interface(MPI) Forum: MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA (1994)
8. E. Mohr, D. A. Kranz, R.H. Halstead Jr.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.* **2**(3) (1991) 264–280
9. S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, C. wen Tseng: UTS: An Unbalanced Tree Search Benchmark
10. C. Pheatt : Intel® Threading Building Blocks. *J. Comput. Small Coll.* **23**(4) (2008) 298–298
11. G. Shekhtman: State Threads for Internet Applications <http://state-threads.sourceforge.net/docs/st.html>
12. K. Taura, K. Tabata, A. Yonezawa: StackThreads/MP: Integrating Futures into Calling Standards. *SIGPLAN Not.* **34**(8) (1999) 60–71
13. R. von Behren, J. Condit, F. Zhou, G. C. Necula, E. Brewer: Capriccio: Scalable Threads for Internet Services. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2003) 268–281
14. K. B. Wheeler, R. C. Murphy, D. Thain: Qthreads: An API for Programming with Millions of Lightweight Threads. In: IPDPS, IEEE (2008) 1–8



## Appendix A: Join implementation with Double-checked Locking Optimization

---

```
// Data structure
typedef struct{
    void *return_value;
    int status;
    thread *waiter;
    mutex_t lock;
    ...
}thread;

// Join operation
void join(thread *th)
{
    void *ret;
    thread *this,*next;
    this=massivethread_self();
    // First check the status without locking
    if (th->status==FREE_READY){
        ret=th->return_value;
        mem_free(th);
        return ret;
    }
    // Then check with locking
    mutex_lock(&th->lock);
    if (th->status==FINISHED || th->status==FREE_READY){
        // Target is already finished
        ret=th->return_value;
        mutex_unlock(&th->lock);
        // Wait for the target ready to be freed
        while (th->status!=FREE_READY){}
        mem_free(th);
        return ret;
    }
    th->waiter=this;
    // Block currently running thread
    next=get_next_thread_from_ready_deque();
    switch_callstack(next);
    // Execute mutex_unlock with following the other context's stack
    // in order to avoid collision of call stack
    mutex_unlock(&th->lock);
    switch_context(next);

    // Execution continues from here:
    mutex_lock(&th->lock);
    ret=th->return_value;
    mutex_unlock(&th->lock);
    // Wait for the target ready to be freed
```

```

    while (th->status!=FREE_READY){}
    mem_free(th);
    return ret;
}

void on_thread_termination(void *retval)
{
    thread *this,*waiter,*next;
    this=massivethread_self();
    mutex_lock(&th->lock);
    // Set return value and status
    this->return_value=retval;
    this->status=FINISHED;
    waiter=this->waiter;
    if (waiter!=NULL){
        // Continue the waiting thread
        switch_callstack(next);
        // Execute mutex_unlock with borrowing the other context's stack
        // in order to avoid collision of call stack
        mutex_unlock(&this->lock);
        // From here thread descriptor is ready to be freed
        this->status=FREE_READY;
        switch_context(waiter);
    }
    else{
        // Execute the other thread in the ready deque
        next=get_next_thread_from_ready_deque();
        switch_callstack(next);
        mutex_unlock(&this->lock);
        this->status=FREE_READY;
        switch_context(next);
    }
}

```

---