Sthreads: Multithreading for SCI clusters

Enno Rehling[1]

[1] Operating systems and distributed systems, University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
{enno@upb.de}

Abstract—

Threads are in widespread use as a model for concurrent programming. In our effort to supply a broad range of tools for the SCI platform, we have created Sthreads, a library that simplifies the adaptation of programs written for the Pthreads library to SCI Cluster hardware. This paper presents the design decisions we made, and a performance evaluation comparing the Sthreads library to Pthreads.

Keywords—**Pthreads, SCI, Cluster computing, Yasmin**

I. Introduction

Threads have become a widespread model for concurrent programming. The conformance of most UNIX and DCE implementations to the POSIX standard has made the POSIX threads standard, commonly known as Pthreads [NIC 96], the de facto standard for writing portable multithreaded applications.

SCI, the Scalable Coherent Interface, is an IEEE Standard [SCI 93] and allows shared-memory programming on large clusters of off-the-shelf PCs, thus eliminating the need for message-passing. Hardware support for shared memory programming simplifies porting of existing applications to SCI clusters, but it also creates the need for new libraries commonly found in conventional SMP systems, e.g. synchronization primitives.

Traditionally, large computing clusters have used MPI or PVM. In order to run an application on such a machine, one had to adapt the software to the respective message passing library, often rewriting major parts of the code, even if it had been written with concurrency in mind. On an SCI cluster, true shared memory programming is possible and efficient, minimizing the overhead for porting of an existing application.

This paper describes the implementation of a threading library for a network of homogenous computers coupled by SCI adapters. It defines a subset of the Pthreads library functions, simplifying the porting of Pthreads-compliant applications to this platform.

Sthreads is one of a number of building blocks for SCI applications being developed at the University of Paderborn. Once completed, the application we are aiming to port is a Java VM [TRA 98] that will treat the entire cluster as a single machine [REH 99] rather than having one VM per node of the cluster.

The paper begins with a brief introduction to the Yasmin library, which offers communication and synchronization for a static number of processes on an SCI Cluster. The library is described in more detail in [TAS 98]. In the next sections, the concepts behind the implementation of the Sthreads library are explained. This currently covers threads and mutexes, but will be extended to condition variables in the near future. The final sections feature a discussion of alternative threading models and performance data.

II. Yasmin

Yasmin (the acronym stands for Yet Another Shared Memory INterface) is a library developed at UPB. It's goal is to simplify SCI programming by providing basic functionality for concurrent programming. Yasmin offers a process-based programming model: At startup, a fixed number of processes per computing node (usually one per CPU) is created.

Local memory from each of the nodes can be mapped into the address space of all processes, providing a uniform address space. Since the memory is mapped to the same address for each process, it is even possible to exchange pointers, something that's altogether impossible in message-passing systems. Yasmin allows dynamic creation and removal of shared segments at runtime.

Because these processes have to synchronize access to the shared segments, Yasmin also offers a number of synchronization primitives: Several different mutex algorithms are implemented, reader/writer locks, as well as signal/wait semantics.

These synchronization operations use a number of algorithms that rely on the number of processes to be fixed. They make internal use of tree structures or arrays, and their efficiency is deeply rooted in knowledge of the number of participants in a synchronization.

Fig. 1 shows the process model of Yasmin. Three processes were created on two hosts. Each has it's own address space, and a shared memory area on the left node is mapped directly into the address spaces of processes 1 and 2, and from a remote mapping into the address space of process three. An access by process 3 would be handled by the SCI hardware, resulting in remote read/write operations on the node that exports the memory.

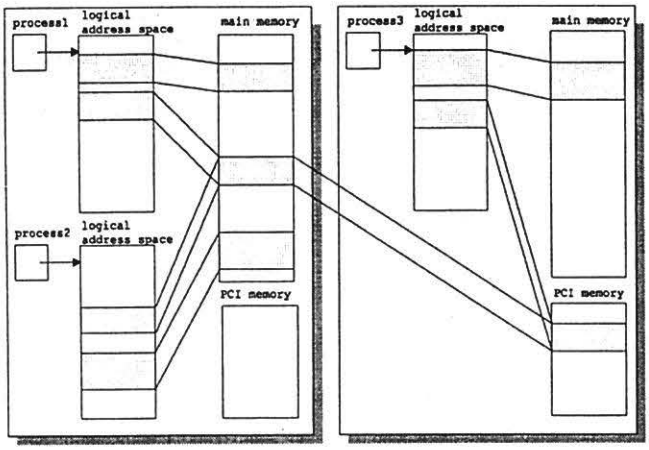

Fig. 1 The process model of Yasmin

III. A LAYERED APPROACH

Using one Yasmin process per thread is clearly not an option. One might postulate a system where a large number of processes is created at startup, and they are left idle until we want to create another thread. However, most of Yasmin's operations require O(log p) time, p being the number of processes, and it's not a good idea to slow down e.g. mutex operations because we create a huge pool of unused processes at runtime.

Thus, Sthreads uses a layered approach. Each node hosts one or more processes, which interact through the Yasmin library. Each process spawns a dynamic number of threads, possibly using the operating system's native thread library or any other such library, even Pthreads. Threads on the same node can use local synchronization primitives to share data between intra-node threads, or use both Yasmin and local operations for inter-node synchronization.

This approach of mixing Yasmin and the operating system's functions is used throughout the Sthreads library. In most cases, functions first synchronize to make sure only one thread on each node enters Yasmin at a given time, and in a second phase communicates with all other nodes through Yasmin. The benefits of this approach are twofold: Not only is the complexity of inter-node communication a function of the number of nodes, as opposed to the number of threads in the system. There is also a number of possible optimizations. For example, condition variables can choose to wake up local threads with a higher priority, eliminating the need for inter-node communication in this special case. See section VII for a discussion on performance.

IV. THREADS

The Sthreads library contains functions to create threads either on the local node or on remote nodes. A thread can access SCI memory together with all the other threads, or share the same address space, unmapped memory can potentially be used by all threads of the same node in order to gain performance. This and other performance considerations will be detailed in section VII.

As for the implementation of the threads, let us begin by looking at what we're given by Yasmin and the OS. In Yasmin, a static number of processes are created during the startup of the program. This could be one process per host or, given that our cluster has SMP nodes, might be one process per processor. However, the number of processes is static, and cannot be changed at runtime.

The operating system provides threads of its own. These can be user-level threads, kernel threads or any variant of the two (see below for a discussion of different threading libraries). For simplicity, we're going to call them *native threads*, as opposed to Yasmin processes.

Thus, we use the aforementioned layered approach. For each node, only one Yasmin process is created (in the case of user-level threads, one process per CPU). This process acts as a daemon thread for the whole Sthreads system. Subsequent creation of threads is done by creating native threads on the target node.

Of course, all calls to Yasmin done inside the Sthreads library have to be protected from access by more than one native thread on the same node. A number of local mutexes are used to ensure this.

A. *Creating threads*

Threads can be created either on the local node or on a remote node. As of the writing of this paper, intelligent assignment of threads to processors is entirely up to the user, but see performance considerations in section VII for a number of possible options.

The thread's resources need to be accessible from all other threads; they must be able to join it, queue it in a number of wait-queues and so forth. Thus, in Sthreads, the `sthread_t` data type, which is the handle given to all operations working on threads points to a data structure in shared memory. It contains a handle for the local thread, the id of the process that it's running in, as well as scheduling information.

Creating a thread in the same process as the creator is straightforward. It's simply a matter of allocating memory for the thread's resources in a shared segment and creating a local thread.

Creating a thread on a remote node is somewhat more difficult. Again, the resources must be allocated, but to increase performance these resources are allocated in a memory segment that was exported by the target node. Parameters for the thread must be accessible from the target node, thus have to be in mapped remote memory, too. Since the program is loaded to the same address, copies of every statically linked function reside at the same address on each

node, a necessity if we want to execute a function on a remote node.

From here on, our problem is that of notifying the target process that it should create the thread. As mentioned in section IV, the main thread on each node acts as a daemon, waiting to perform request from remote nodes. One such request is creation of a thread; a small message stating the location of the thread resources triggers the daemon and makes it start a new thread. Again, performance is a critical issue, and we make a few observations in section VII regarding the use of interrupts versus polling.

V. MUTEXES

As mentioned in section II, mutex operations in Yasmin rely on knowing the number of processes involved at the time a mutex is created. The data structures used for implementing the lock and unlock operations do not allow adding or removing of processes. They are also not threadsafe, making it impossible to share the same mutex between two threads of the same Yasmin process. Thus, we are unable to use Yasmin's mutexes directly for synchronization in the Sthreads library.

Also, since a Yasmin mutex is local data, e.g. the mutex object cannot be exchanged even among hosts, a different representation has to be found for the implementation of Sthreads mutexes.

On the other hand, native mutexes are only valid in the context of processes running on the node where they are created. A native mutex created on host A cannot be used on host B. Obviously, native mutexes cannot be used for synchronization either.

Instead, Sthreads mutexes use a mixed approach: They rely on native mutexes for synchronization among Sthreads running on the same host, and use yasmin's mutexes for inter-node synchronization.

Locking and unlocking Sthreads mutexes is fairly simple:

```
sthread_lock(sthread_mutex_t lock)
{
    native_mutex_lock(lock.native_mutex)
    yasmin_mutex_lock(lock.yasmin_mutex)
}

sthread_unlock(sthread_mutex_t lock)
{
    native_mutex_unlock(lock.native_mutex)
    yasmin_mutex_unlock(lock.yasmin_mutex)
}
```

The complicated part about these mixed mutexes is creation. Because in Pthreads every thread at any time can create a new mutex simply by calling the `pthread_mutex_create` function, the operation may clearly not be a synchronization point. Local mutexes for all other nodes must either be created at a later time, or they must have been created previously.

In Sthreads, the solution was to internally allocate mutexes beforehand. Apart from wasting a few resources, this is perfectly okay, and we can do so at a synchronization point. To globally identify a Sthreads mutex between multiple Sthreads, an integer is used. Locally, the integer functions as an index to an array of Yasmin mutex and native mutex.

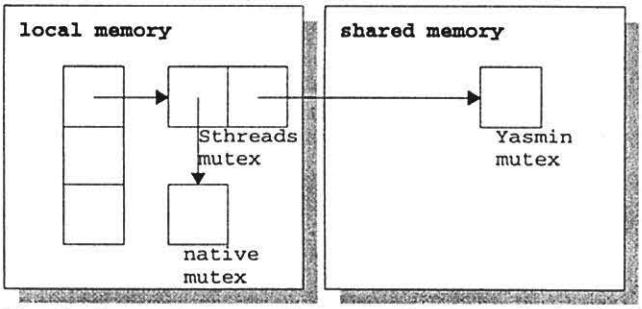

Fig. 2 shows the general layout: the array is in local memory, and using the index that `sthread_mutex_t` represents, we can look up the tuple of native mutex and Yasmin mutex. Only after locking the native mutex, may the Yasmin mutex situated in shared memory be accessed.

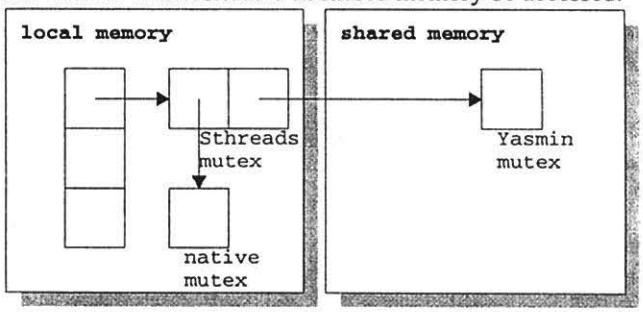

Fig. 2: An Sthreads mutex consists of a native mutex and a Yasmin mutex

VI. ALTERNATIVE THREADING LIBRARIES

Today, most operating systems supply Pthreads. This includes Solaris and Linux, the two operating Systems that run on the PSC [PC2 99]. However, we currently support an alternative threading library, a user-level thread implementation taken from the Kaffe virtual machine[TRA 98].

There are both benefits and drawbacks of this: The major drawback is that user-level threads will not make use of SMP nodes. On an SMP node, running multiple Yasmin processes will let us make use of more than one processor. However, although communication among local processes is a lot faster than communication between processes on different nodes, native threads will offer better performance.

On a single-processor node, user-level threads are usually faster than native threads, because they do not suffer from costly kernel entries. The other benefit apart from speed is the ability to better debug user-level threads,

and the fact that user-level threads and mutexes might be coupled closer to the Sthreads library, as explained in Section VII.

VII. PERFORMANCE

In order to evaluate the performance of Sthreads, a number of tests were performed. To show the efficiency of the threads, we present figures for thread creation and finalization, as well as for a system under heavy load.

Each test was done on one or more Dual-Pentium II/400 nodes running Linux 2.2.6, and figures for Sthreads are compared to Pthreads performance on a single Node.

B. *Thread creation and termination*

The time required to create *n* threads and wait for their termination is shown in . We found that the predominant factor in these operations is the efficiency of the memory management system, and the Sthreads implementation suffers from Yasmin's rather simple implementation of it. Improving it will be a priority issue for the next version of Yasmin. Nonetheless, the performance compared to Pthreads is already reasonably good.

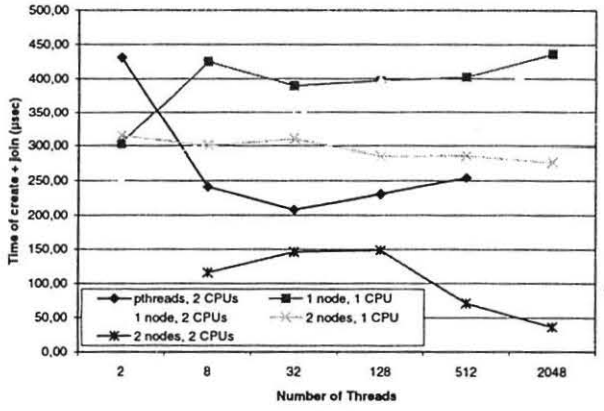

Fig. 3: Execution time of `sthread_create` *and* `sthread_join` *compared to Pthreads*

C. *Thread scheduling*

In this test, each thread was given the task of sorting a 32 KB array of integers. The time it took all *n* threads to terminate was measured, and it gives an indication of the overhead introduced through thread scheduling in Sthreads.

The graph below shows the relative execution time compared to a sequential program. In general, Sthreads do not perform as well as Pthreads for loads greater than 64, but the overhead never exceeds 10%.

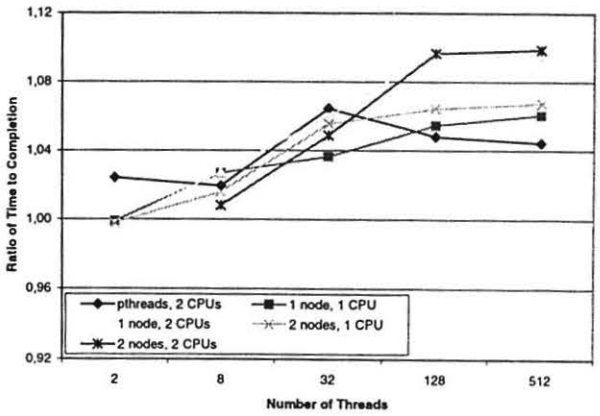

Fig. 4: Running time under different load relative to sequential time

D. *Mutex operations*

The efficiency of Sthreads mutex operations depends mostly on the underlying mutex functions. Yasmin offers a range of mutex algorithms. They are discussed in more detail in [TAS 98] and we include Fig. 5 from this paper to give a brief overview of their relative performance.

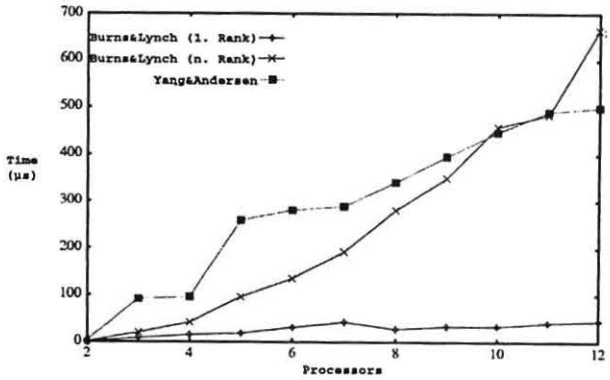

Fig. 5: Performance of the three optional semaphore algorithms in Yasmin

E. *Future Improvements*

There are several options for increasing the performance of Sthreads. As mentioned before, placement of threads is currently in the hands of the application, and there is no support from the library to help in the decision. By keeping track of the load on each node, intelligent placement of threads might considerably improve application performance.

Another possible improvement would be the exploitation of thread locality. A thread that releases a mutex object or signals a condition variable might decide to hand it to a waiting thread which is on the same node. In

this case, only local operations would be required, where currently Yasmin locks are used. Furthermore, by relaxing the scheduling model, local threads could always be preferred to threads on remote nodes, requiring some strategy to prevent starvation.

Finally, recent discussions in the SCI community have shown that there is no clear-cut answer to the question of whether to use polling or SCI interrupts to notify processes. Clearly, polling is a bad option to use for the guardian thread mentioned in section IV, but other problems might be solved more efficiently by polling or mixed strategies.

VIII. References

[DOL 96] Dolphin: *The Dolphin SCI Interconnect*. White Paper, Dolphin Interconnect Solutions, Olaf Helsets Vei 6, 0621 Oslo, Norway, http://www.dolphinics.com/.

[NIC 96] B. Nichols, D. Buttlar: *Pthreads Programming*, O'Reilly & Associates, Inc, September 1996.

[OMA 98] K. Omang: *Performance of a Cluster of PCI Based UltraSparc Workstations Interconnected with SCI.* In Proceedings of Network-Based Parallel Computing, Communication, Architecture, and Applications, CANPC'98, Las Vegas, Nevada, Jan/Feb 1998, Lecture Notes in Computer Science no.1362, pages 232-246.

[PC2 99] PC^2 Homepage: *The PSC Primergy High Scalable Server*, http://www.upb.de/pc2/systems/psc/

[REH 98] E. Rehling, R. Butenuth, H.-U. Heiß: *Project Arminius Homepage*, http://www.upb.de/cs/heiss/arminius/

[REH 99] E. Rehling, R. Butenuth: *Project Arminius*. In Proceedings of the 2nd Workshop on Cluster Computing, Karlsruhe 1999.

[RYA 97] Stein Jørgen Ryan: *The Design and Implementation of a Portable Driver for Shared Memory Cluster Adapters*. Research Report no.255, Department of Informatics, University of Oslo, December 1997.

[SCA 99] Scali Computer Homepage: http://www.scali.com/

[SCI 93] IEEE: *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE standard 1596-1992, New York, 1993.

[TAS 98] H. Taşkın: *Synchronizationsoperationen für gemeinsamen Speicher in SCI-Clustern*, Diploma Thesis at the University of Paderborn, 1998

[TRA 98] Transvirtual Technologies: *The Kaffe Open Source VM*, http://www.transvirtual.com/