Thread-based Communication Environment

Janusz Niemiec, Wojtek Furmanski, Geoffrey C. Fox

Northeast Parallel Architectures Center
111 College Place
Syracuse University
Syracuse, New York 13244–4100
[janusz,furm,gcf]@npac.syr.edu

August 21, 1995

Abstract

The light-weight multihreading is becoming recognized as the most consistent and uniform method in supporting concurrency and overlapping between computation and communication. So far, however, parallel communication systems lack threading capabilities. This paper introduces Thread-based Communication Environment (TCE), which provides an efficient integration between message passing and multithreading. Our library is constructed as a portable layer for distributed memory, heterogenous parallel and distributed systems. It ensures interoperability between applications structured accordingly to data or functional parallel paradigms, executed on different platforms. The paper examines the design and implementation issues concerning the development of the library. It describes in detail the TCE interface calls and illustrates them by excerpts from real codes. In conclusion, we present timing data of the TCE implementation and performance gains for a typical parallel application obtained while using multiple TCE threads.

1 Introduction

Multithreading support has become increasingly important for many classes of applications. Light-weight threads can benefit any application which contains a complex control flow, asynchronous actions, needs to hide IO latencies, performs dynamic load balancing, or wants to overlap computation with communication. However, among HPC architectures, the multithreaded systems utilized sufficiently only the shared memory platform because the standard functionality of a thread package naturally supports multiprocessors. To employ the HPC distributed memory architectures, thread packages have to be enriched by an efficient communication mechanism. This was our primary motivation to construct the Thread-based Communication Environment (TCE).

Thread-based Communication Environment provides a uniform communication platform across Internet-wide, heterogenous architectures. A network of TCE daemons manages different resources, starts TCE processes and routes messages between nodes when necessary. TCE processes are multithreaded and communicate among themselves via message passing. The thread scheduling is prioritized, confined to process boundaries and can be either preemptive or non-preemptive. Communication between TCE processes is based on two abstract communication objects — ports and channels. Ports represent processes' points of contact, allowing a construction of higher level communication links — channels. Channels provide provisions for efficient and flexible inter-thread interactions, which refer exclusively to two peer processes. They can also be grouped into channel sets to define a group of participants for collective communications.

The TCE library can be used as an HPC compiler target or as a runtime library for message passing oriented applications.

2 Background

2.1 Threaded Systems

Threads gained their importance when it became apparent that standard processes, supported by Operating System can be too costly in terms of context switching and inter-process communication. Swapping processes' contexts is a complex operation that requires the exchange of not only the contents of registers but also Operating System specifics like stacks, page tables, signal vectors. Often, all caches have to be purged too. In contrast, the switching of threads contexts takes only the interchange of machine registers. Correspondingly, inter-process communication for regular processes is related with crossing process boundaries, which has to be performed using the system supplied mechanisms such as semaphores, shared memory segments, pipes or sockets. Those operations have to be carried out through expensive system calls and their completion often results in an additional, heavy-weight context switch. Interaction between threads, on the other hand, can be performed cheaply by accessing common data structures.

A quantitative comparison discloses that for contemporary RISC processors a context switch between regular processes is in a millisecond range. Whereas, a context swap among threads is in a microsecond range [23]. Inter-process communication latencies start from tens of microseconds [34], while accessing a single data element consumes from a couple to hundreds of nanoseconds[1].

2.1.1 Kernel-level Threads

Contemporary Operating Systems offer a multithreading support managed by the Operating System kernels. Mach [1], Amoeba [29] or Windows NT [6] are examples of this tendency. Their multithreading support can be very elaborate (Mach and Windows NT) or rather modest (Amoeba), but in all cases threads are scheduled preemptively by the kernel and synchronization is done through mutexes and semaphores. An Operating System schedules threads the same way it schedules processes; a thread which cannot complete its system call is blocked, and a thread which exhausts its time slice is preempted. The next thread selected to run can belong to the same process as the blocked or preempted thread. This is the advantage of the kernel-level threads — a blocking system call will only block the thread that issued that call, not the whole process. The disadvantage of the kernel-level threads manifests itself in high costs of context switching and thread management. During a context switch the control has to pass the protected boundary between the user and kernel address spaces. In addition, it is necessary to preserve other information such as signal and descriptor tables for systems which allow those to be private to a thread. A thread can be executed in the kernel and the user mode, so two stacks per thread have to be allocated, maintained, and preserved. Therefore, threads managed by the Operating System kernels are often called medium-weight threads.

2.1.2 User-level Threads

The user-level threads, on the other hand, are very efficient when the context switching cost is concerned. All necessary operations can be executed in the user space, without any kernel assistance. It comes at the expense of often limited functionality, because the whole thread management must be done exclusively within a thread package. Another disadvantage is that a thread blocked in the kernel will block all other threads belonging to the process. Therefore, either thread packages have to provide their own versions of blocking system calls by writing a layer on top of the non-blocking calls ("jackets"), or an application has to have

[1] On the DEC 300 AXP Model 400 workstation reading a word item requires 7.5ns, 75ns and 203ns when the item is in the primary cache, secondary cache or DRAM respectively [7].

access to scheduling primitives and invoke non-blocking system calls itself. The first option is obviously more plausible, as it releases an application from dealing with many messy details, but imposes some new difficulties for a package implementation.

There are many different user-level thread packages. The Light-Weight Processes for SunOS [31], C-Threads for Mach [33], the OSF DCE threads [30] and the POSIX, pthreads package [28] are among the best known. The common functionality found in those libraries contains thread creation, destruction, scheduling and synchronization. More specific features, such as a preferable scheduling policy, preemptiveness, priorities, control over stacks, and signal handling, are only present in some of them. Dealing with blocking system calls is entirely left to the programmer.

Some approaches try to combine advantages of both, the kernel- and the user-level threads. OThreads [9], a user-level thread package, eliminates the problem of blocking in the kernel by mapping user-level threads onto kernel threads. The scheduler in the user space activates a kernel thread when a blocking is anticipated and stops it when a resumed kernel thread adds to an excessive number of running kernel threads. However, this solution is only partial because it is assumed that blocking can be anticipated by an application. A complete solution is presented by the Psyche Operating System [26], which notifies a process about a possible blocking or a preemption using software interrupts. A scheduler in the user space can thus take appropriate decisions, such as blocking a thread and starting a new one or preparing the process for the preemption. Unfortunately this kind of support requires significant changes in the Operating System kernels.

As it was mentioned earlier, the fundamental benefit of the user-level threads is their efficiency. However, a fast implementation comes at the expense of functionally and a rich functionality endangers performance. To let the developer decide about a desired balance, David Keppel constructed a minimal thread package — QuickThreads [23]. This package provides only basic primitives for thread creation, destruction and switching. Everything else, including scheduling, is left to the developer. This allowed to construct a very fast thread package [20,23], with an additional benefit of high portability. The QuickThread library has been used as the basic thread layer in the development of TCE.

2.2 Multithreaded Communication Systems

Multithreading deals in a natural way with shared memory multiprocessors. However, when multiple processors have physically disjoint memory banks, additional communication mechanisms are needed to permit inter-process interactions.

2.2.1 Solutions for Distributed Computing

The previously discussed multithreaded Operating Systems allow inter-process communication via abstract communication objects called ports. These abstract objects perform two important functions — security and location transparency in addressing. This is especially important in distributed environments, where participating peer processes cannot be trusted in general, and processes or services can be migrated or replicated to improve availability and fault tolerance. Mach and Amoeba are the most consistent in using ports as the communication mechanisms. Ports in Mach allow a variety of styles of communication, including asynchronous and synchronous message passing, RPC, byte streams and other. The communication is buffered and the receiving port acts like a mailbox. In contrast, Amoeba utilizes ports only as a way to carry through RPC service. Both server's and client's requests are blocking. The Mach port-based solution is much more elaborate and complete, since port rights can be migrated to other processes, ports can be grouped in a port set and all system services are accessible via operations on ports.

2.2.2 Solutions for Parallel Computing

Multithreading communication systems were not present in the distributed memory, parallel architectures for a long time. A modest operating system support and a conviction that a context switching among threads

was too expensive contributed to that situation. However, the recent research in the area of the user-level, light-weight multithreading seems to be successful in removing those obstacles. Moreover, a large number of parallel applications started requesting multiple threads of execution either at a language runtime level [3,16,22] or directly at the application level [19,11,15]. A simplification of complex control flows, clean semantics for asynchronous operations, and a better responsiveness are among the main reasons for this demand.

• Historically, NewThreads, an object-oriented runtime library [11], was the first among multithreaded communication packages. NewThreads furnishes a non-preemptive multithreading and a port-based, blocking communication. Ports have globally unique identifiers. They can be associated with literal names and then looked up through a global naming service. The package stub compiler, generating the client and server stubs, supplies a support for the RPC communication.

• Nexus [12] runtime interface has been developed to support interoperability between different runtime layers of parallel languages. Nexus's threads are only visible within a single address space (Context), and they can access data within remote contexts using a {Node,Context,Address} triplet. The triplet is called Global Pointer. A Remote Service Request can be issued to access remote data, provided with a Global Pointer, handler identifier, and a message buffer. The remote service invocation is asynchronous, and the appropriate handlers can be either threaded or represented as ordinary functions. Currently, every remote invocation has to be preceded by a handler name to a handler identifier lookup. Threads scheduling can be either preemptive or non-preemptive.

• The Chant "talking threads" runtime library provides point-to-point communication primitives and remote service invocations. The point-to-point method addresses receiving threads by their globally unique identifiers. Before an actual message passing can take place, both the sending and the receiving threads have to agree on that transfer. Thus, at the receiving side no buffering is ever required. The remote service is based on the point-to-point method, which first delivers a request to a dedicated "listening" thread, and then forwards it to the addressed handling thread. Instead of offering its own, customized, thread and communication interface, Chant provides interfaces based on the Posix pthreads and MPI communication library. The actual implementation platform can be, however different, depending upon availability of those libraries on a particular architecture. Unlike NewThreads, communication calls can be either non-blocking or blocking. In the former case, only the thread that issued that call is blocked. Threads in Chant are globally visible, and all operations on them can be applied locally or remotely. They can be grouped into ropes; sets of threads on which collective communications are defined. The Chant scheduling is non-preemptive.

• The TCE library is closest in spirit to NewThreads. Likewise, the TCE communication is based on abstract objects and all communication calls have blocking semantics. Ports can be obtained by the naming service and threads can only be visible within a process address space. However, the TCE communication is much more powerful, allowing customizing of channel behaviors, passing channels over to other processes, and constructing collective channel sets. In contrast to NewThreads, TCE ports are used only as points of contact, necessary for establishing channel connections.

TCE offers the most complete support for the data parallel SPMD paradigm through a rich set of collective communications defined over channel sets. Among all discussed previously libraries only Chant furnishes similar support in the form of ropes. In TCE, an application can thoroughly control various aspects of scheduling, including preemptions, clock intervals and thread priorities. This kind of functionality is indispensable for irregular, dynamic in nature applications as well as multiuser servers. Only Nexus provides a preemptive execution, but its aspects cannot be customized. None of the 3 libraries, however, make any provision for location transparent, client-server dynamic binding, essential for building resident parallel services. TCE makes it possible by registering globally visible ports, which then can be retrieved using a network of daemons. The urgent need for this capabilities may not be apparent now, but we believe it will be in a rising demand as the current World-Wide Web technologies approach its limits in performance and scalability.

3 Design Postulates

The TCE library has been designed to provide support for parallel languages runtimes as well as for end users on a variety of MIMD HPC platforms. We will now present briefly the goals of the TCE design, examining both our choices and compromises.

3.1 Simplicity

Programming with threads often introduces new difficulties, which are likely to occur in more complex applications. Simplicity of the supporting environment is thus very desirable, enforcing a strong programming discipline and minimizing number of places where errors can be incurred. This reasoning guided us in choosing a TCE specific interface and implementation for multithreading and communication. An alternative solution, adopting already developed thread and communication packages and coupling them together, would produce an environment which is too heavy or too inefficient. For example, simply merging the MPI [27] message passing standard with the pthreads POSIX thread standard would result in a library with approximately 200 calls. A number large enough to make such a library hard to comprehend. Moreover, many of those calls are not strictly necessary.

• In order to overlap computation with communication, many message passing systems provide non-blocking (asynchronous) sends and receives, often complemented by interrupt driven handlers [21,32,34]. Using multiple threads, all non-blocking calls can be eliminated because a thread that issued a call which cannot be completed immediately will be blocked, awaiting for the completion. In the meantime, other ready threads can proceed with their activities, therefore making the overlap possible. It has to be stressed out that this method not only retains a clean and familiar, blocking semantics but also makes programs easier to write, understand and debug. As the result, the TCE implementation reduces the number of point-to-point primitives to 2 — send and receive.

• An application or a runtime system is released from scheduling TCE threads. All scheduling decisions are taken internally by the default scheduler. The scheduling policy is based on priorities and is fixed. Nevertheless, the TCE scheduler can be instructed about thread priorities as well as about the preemption mode and threads' time slices.

• All operations on threads are confined to threads local to a process. This way we have achieved a significant reduction in the implementation complexity. Furthermore, it generalizes an interoperability between different processes, permitting them to be single threaded. The other alternative, chosen by Chant for instance, allows threads to be created and synchronized across processes. If such a mode is desirable, it can be carried out by an application itself; a "listening" thread may wait for requests from other processes' threads and apply them to local threads.

• Collective communications are performed over a defined group of participating sites. A defined group can be either static or dynamic. A dynamic group allows participants to join the group and freely leave it, while the static nature demands that all members of the group have to participate throughout the group lifetime. The TCE channel sets form only static groups. Similarly to the previous example, by taking advantage of multiple threads the dynamic nature can be implemented on top of the provided functionality. For example, each member may create a thread awaiting for a request to close the current group and create a new, extended or shrunk, one.

• To reduce the number of interface calls we made the interface functions as polymorphic as possible. For instance, abstract communication objects such as ports and channels can be transferred by send/receive, the same primitives which support sending and receiving data buffers. The channel abstraction also helps here; instead of supplying an additional set of operators, different modes for buffering and handshaking have been expressed by imposing appropriate characteristics upon channels.

3.2 Efficiency

The TCE library is targeted at HPC architectures, therefore it is extremely important that they can be utilized efficiently.

• The context switching cost is a dominant concern for all developers of thread packages. We have chosen a minimalistic and thus fairly fast package — QuickThreads as the basic layer. On top of it an efficient, prioritized scheduler with preemptive capabilities has been added.

• Thread scheduling is closely integrated with the communication subsystem. Since blocking on communication is frequent, the "floating" scheduler concept has been adopted. Instead of switching to a scheduler thread and then to a newly selected thread, the TCE scheduler executes itself directly on threads' stacks. This solution allows for saving of one context switch. Likewise, the communication subsystem does not have its independent threads but all network functions proceed using stacks of application threads. The system, though, has to ensure that all thread stacks are enlarged by the amount needed for the execution of the TCE kernel functions.

• On HPC distributed memory machines, buffering and handshaking determine the efficiency of communication operations [8,10]. TCE channels can be configured to perform buffered receives, non-buffered receives which requires an extra handshake, and non-buffered receives without any handshake which however may lead to a message loss. It is up to the programmer to decide about a method that suits his program needs best.

• There are places in the TCE implementation which cannot be implemented efficiently for all possible cases. In those situations we tried to optimize only the paths which are most likely to be chosen frequently.

— A widely used approach in speeding up a process of new objects' creation is to keep an internal cache of objects ready to use immediately. If a cache is empty, a regular creation will take place. TCE uses that principle in the case of creating new threads and message headers. The difference in performance can be dramatic, for example, the creation of a new thread that is found in the cache is faster than a context switch. Normally, a creation is from ten to hundred times more expensive than a context swap [20]. Since caching reduces the amount of available memory, the number of cached items is under application control.

— TCE is supposed to provide Internet-wide interoperability between processes running on different HPC architectures. To implement this, we needed to use message headers, which provide the TCE network system with various information regarding the sending and receiving ends. Message headers generate negligible overhead for message passing via the TCP/IP protocol. However, they can incur a significant performance loss for distributed memory parallel machines. Communication on such systems is datagram oriented, and does not allow to send or receive data from/to different memory locations. Thus, either two extra memory copies have to be introduced, or a header and a buffer must be transferred in separate communication steps. To cope with the problem, we again optimized the most frequent case. All standard sends and receives which do not cross machine (partition) boundaries use the message tag field to address the recipient. All other messages have the regular, header plus buffer, structure.

3.3 Portability

Portability is an issue of paramount importance, particularly significant for the fast changing HPC platforms. The TCE multithreading and communication interfaces are machine and Operating System independent, thus an application is shielded from system incompatibilities. On systems which do not provide required support for certain functionality, such as signals or interrupts for preemptive scheduling for example, appropriate interface calls are redefined to void operations. Therefore, an application can still be compiled and executed. Since TCE is based on the user-level multithreading, porting to new architectures is relatively simple, and requires mainly developing short assembly routines for new processors.

3.4 Interoperability

While designing the TCE library, two important goals have been stated — the system should support interoperability between different programming models as well as various computer architectures.

• TCE supports both data parallel and functional parallel paradigms, and they both can be mixed freely even within the same application. It has been accomplished by implementing the host-node programming style, according to which an application can have a complete control over how and where its processes are placed. Since most of parallel applications exercise the data parallel paradigm, our environment furnishes a special support for it by providing a rich set of collective communications.

• Applications which use the TCE package can take advantage of registered ports to implement the distributed computing, client-server model. By having the distributed and parallel models supported, it is possible to develop much more complex systems such as parallel servers or clients.

• Using our package, it is possible to take advantage of various, heterogenous computing resources. Except SIMD machines, all other low- and high-end architectures are covered. To provide a dynamic access to all currently available resources, TCE uses a network of daemons. Daemons are placed on all nodes, which are both listed in configuration files and are loosly-coupled with other nodes. A tightly-coupled, parallel system has only one daemon placed at its front-end.

• Applications running on machines with different data formats will still interact properly. Each TCE communication call requires the specification of the format of transmitted data and the translation will be done automatically when needed.

3.5 Abstracting Physical Resources

In constructing TCE we wanted to avoid addressing either hardware or software resources by their physical identifiers. Abstracting them we could obtain better portability and location transparency. But while doing this we had to be careful not to obstruct details which applications might choose to care about.

• In order to place its processes, an application has to perform mapping between them and previously allocated computational nodes. Those nodes are represented by abstract objects — it is possible to either request them directly by their Internet addresses and processor numbers or let the TCE runtime to choose appropriate nodes arbitrarily. The first way permits to take advantage of particular machines which an application plans to utilize. The second one automatically chooses the lightest loaded nodes in an easy, transparent manner.

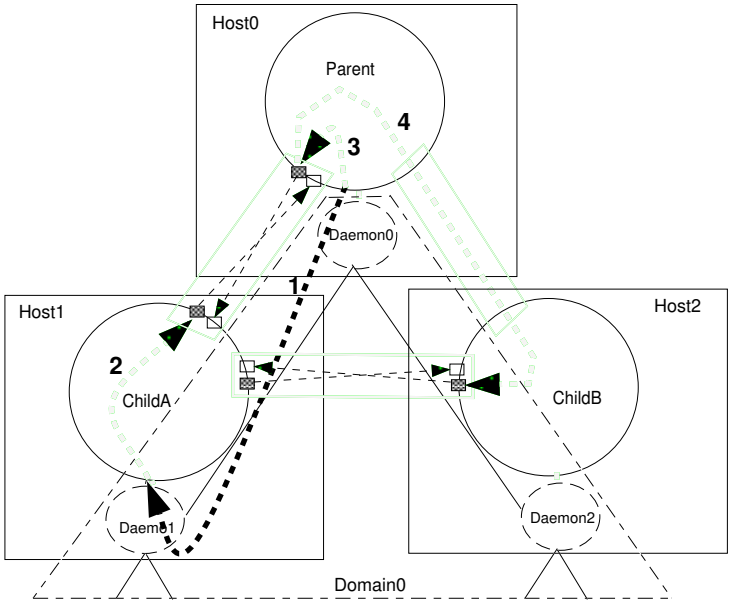
• Some message passing libraries, such as Express [32] and P4 [2] address message recipients by their node numbers. MPI [18] uses processes' ranks and PVM [17] uses tasks' identifiers for this purpose. For a multithreaded environment those methods are not "precise" enough. We could choose to use thread identifiers for this , but it has already been decided that threads are only visible within a process. Instead, following some Operating System and language specific solutions [1,6,13,29], we have adopted abstract communications objects, ports and channels, to carry out inter-process interactions. Ports represent processes' points of contact and are used to create communication links — channels. Depending on how an application defines ports and channels in terms of on- and off-stack variables, they will be either accessible by all processes' threads or be local within a thread context. In our opinion, addressing message recipients via communication objects is more flexible than sending them to particular threads. New threads are often spawned as "helpers", which do their job and then exit. Such threads, couldn't receive arbitrary messages or they would have to remain permanently active.

• On some distributed memory parallel architectures the most efficient communication takes place when performed between neighboring nodes. Programs which are capable of taking advantage of that can still run efficiently under TCE library. An architecture name and a relative process rank are obtainable, thus an optimal communication for a given machine can be determined.

4 TCE Implementation

The Thread-based Communication Environment consists of three layers. The first, bottom layer is concerned about managing various heterogeneous resources and ensuring proper interactions between them. This layer is not directly visible to applications. The next, upper layer deals with presenting parallel/distributed resources in an abstract way and placing user processes on them. The third layer deals with threads and communication via ports, channels and channel sets.

In Figure 1 a general illustration of interactions between these layers has been given. The next sections provide thorough description of the TCE daemons, process placement, multithreading and communication.



1) The parent process spawns a new process on the Host1 node. The creation request is presented to the Daemon0 daemon which then forwards to Daemon1 on Host1. Daemon1 starts the ChildA process.
2) The ChildA process connects to its local daemon and receives a sending port to the parent's receiving port.
3) As the result of the creation, parent gets sending rights to its child's receiving port. Now the parent and the child can establish a channel, using their sending and receiving ports.
The same operations are repeated to create a channel between the parent and its second child (ChildA on the Host2 host).
4) Parent sends to the second child the sending port to the first child's receiving port. Identical operation is performed to install the sending port to the second child for the ChildA process. Now both children are in position to establish their direct link.

Figure 1: This diagram illustrates a construction of a network of TCE processes and communication links between them.

4.1 Managing heterogeneous resources (first layer)

In order to efficiently manage different heterogeneous resources available on a network, TCE builds its own representation of those. Specially constructed processes (commonly called daemons) are placed on appropriate network nodes. TCE daemons do not usually participate in the communication between application processes. Their main task is to manage computer resources (monitoring their availability, load, etc.) and to start application processes on appropriate nodes. Because of their presence, applications can place their parts on most lightly loaded nodes in a completely transparent fashion. Furthermore, programs don't have to be aware of how many network nodes are out there for their disposal or what the machine addresses are. TCE daemons deal with those problems automatically when provided with a requested architecture type.

Daemons will not be involved in message passing between processes, unless the direct, process-to-process communication link cannot be established. At these instances, the inter-process interactions are carried out through message forwarding, performed by the daemon layer.

All loosly-coupled nodes have their daemons placed on them, one per node. For tightly-coupled nodes (distributed memory multiprocessors) there is only one daemon per machine, situated at a multiprocessor front-end. There are two important reasons for this distinction. Firstly, multicomputer nodes in almost all cases lack multitasking capabilities, therefore a daemon process could not be running concurrently with an application process. Secondly, all multicomputer nodes within a partition are virtually fully connected thus no routing is needed. Communication outside partitions or machine boundaries can be handled by the front-end process, therefor the daemon is placed there.

Daemons are grouped into domains, which are linked together via a binary sorted tree structure. The same tree structure is used to connect daemons within a domain. A domain place in the global hierarchy as well as a daemon location inside a domain are depicted by their ID numbers. The numbers, together with a process identifier, form a composite address that uniquely defines a TCE process and its location. Because of the binary tree structure, any message carrying the composite address can be automatically routed by daemons, and can cross any number of domains before reaching the destination process. No global knowledge of the network is required, and the message will be forwarded via the shortest path, even when a final destination is not known to the forwarding daemons.

Domains are created by the user and facilitate two goals: maintaining efficient communication within a given domain and building a hierarchical structure of multiple domains. Domains can also be used to impose some restrictions regarding access rights and authentications, however currently TCE does not address any of the security issues. A domain is spanned over a network of workstations or a parallel machine. In the last case, a single multicomputer corresponds to single domain. In the workstation case, domains ordinarily contain machines which are placed near by on a network and have access to the same file system.

A particular domain is built as a set of "regular" daemons and a coordinating daemon(s). For every domain, there is a configuration file which specifies which daemon(s) perform(s) coordinating tasks for this domain as well as addresses of machines which are coordinating daemons for the domain on top and two domains below (tree structure). There can be one or more than one coordinator for each domain. The configuration file may also contain information on domains not directly connected with the current domain. All this redundancy provides a better fault tolerance of the structure.

As we mentioned it earlier, each daemon is a part of a local, restricted to that domain, binary sorted tree. This local hierarchy helps in domains with a large number of nodes. Generally, all Operating Systems limit a number of opened descriptors. A system without forwarding capabilities is then naturally limited in its size. Reserving a small number of descriptors for the tree-like connections (3 for ordinary daemons and 5 for coordinators) guarantees that all daemons are indirectly linked via tree paths A daemon is free to open any number of direct connections to others. However, the tree-like connections have to be established first, thus any communication that cannot be performed using direct daemon-to-daemon links can be carried out through forwarding. The same rules refer to workstation-based application processes. They have to connect to its local daemon first, and after that any other connections can be opened.

Any daemon or an application process can be identified by its private number. The first part of the address identifies a daemon domain (16 bits). The last bit is reserved for a flag which shows whether a domain covers a parallel machine or a distributed cluster. A node (daemon) specific ID is given in the second half of the address (16 bits too). Such a composite address enables locating domains as well as daemons within a domain. The system is able to encompass up to 32K domains and 64K daemons inside a domain. Since there can be more than one process controlled by a given daemon, those processes are independently numbered by an additional 8–bit identifier.

4.2 Process Allocation and Management (second layer)

An application that uses the TCE library has to allocate computer resources explicitly. Parallel or distributed programs must have a main process which then allocates suitable nodes and places processes on them. It is similar to the Express host-node mode, the P4 master-slave style or the PVM way of loading tasks onto nodes.

The main responsibility of this layer is to assist the programmer in choosing computational nodes and starting processes on them. Our goal hare was to find a desired balance between transparency and flexibility in allocating resources. For some applications, it is not of the primary importance on which nodes their processes are started or how many of them are assigned to the same nodes, as long as they can cooperate among themselves. Thus the location transparency feature can be highly desirable. For other application, however, it could be disastrous to use an Ethernet-based cluster of 16 RS6000 machines, instead of 256 node SP-2. Therefore, we provided two modes in acquiring nodes; they can either be requested by their physical addresses (Internet address + processor number) or choose automatically through the daemon layer. In the later case, an application can give a hint regarding what type of architecture it expects to get (see Figure 2). If no suggestion is provided, the TCE bottom layer will assign the architecture on which the master process is being executed.

```
/* Program asks the system to allocate  NSlaves nodes of type Alpha
   TCE runtime will choose those automatically .*/
#define NN 3
int NSlaves = NN - 1;
...
TCE_Node *Nodes[NN - 1];


int N;
...
  if((N = tce_nodes(Nodes,(char **)0,TCE_ARCH_Alpha,NSlaves)) < NSlaves)
...
```

Figure 2: Transparent allocation of computational nodes

When the appropriate nodes have been allocated, they are returned to the requesting process as an array of abstract objects called Nodes. The main process can then map its additional processes onto them (as shown in Figure 3). If a desired set of nodes is wrongly addressed or not accessible, the tce_nodes function will place null pointers in corresponding slots of the returned array. Therefore, by checking the returned value of tce_nodes and by examining the content of the array of nodes it is possible to determine for which nodes the allocation failed.

Similar steps can be applied in the case of starting processes on nodes. The tce_exec function takes the vector of allocated nodes, a name of an executable to be started, an array of receiving ports, and a number that specifies how many copies of that executable should be spawned. The function returns a number of actually created processes and a vector of sending ports. If the spawning failed for a particular node, a corresponding sending port pointer is assigned to null.

It should be pointed out, that for some parallel architectures the number of allocated nodes within a partition must be a predefined quantity (a power of 2 for hypercubes for example). If an application requests a number which does match one of the predefined sizes, the TCE runtime will try to reserve a larger partition, closest in size to the requested figure. Therefore, some parallel partitions may not be fully utilized.

```
/* Allocates NSlaves nodes of type Alpha given in NodesNames[]
   Allocated nodes returned in Nodes[]
   tce_exec places n copies of SlaveName program on allocated nodes */
#define SlaveName "/home/users/janusz/slave"
#define NN 3
int NSlaves = NN - 1;
...
char *NodesNames[NN - 1] = {"Node1:1","Node2:1"};
TCE_Node *Nodes[NN - 1];
TCE_Port *RemotePorts[NN - 1];
TCE_Port *LocalPorts[NN - 1];

int N;
...
  if((N = tce_nodes(Nodes,NodesNames,TCE_ARCH_Alpha,NSlaves)) < NSlaves)
    tce_exit(1);
  if((N = tce_exec(RemotePorts,Nodes,SlaveName,LocalPorts,N)) < N)
    tce_exit(1);
...
```

Figure 3: An explicit node allocation and mapping processes onto them.

4.3 Multithreading and Communication (third layer)

4.3.1 Multithreading

The implementation of multithreading furnishes the user-space, prioritized threads, scheduled in a preemptive or non-preemptive fashion and synchronized by semaphores. A single TCE process may contain any number of light-weight threads, and the number is only limited by the amount of available memory.

The fundamental design issue was either the TCE kernel should be implemented using separate threads or as non-threaded, regular functions. We have chosen the second alternative, which causes all TCE kernel functions to be executed application threads' stacks. Although this method requires more careful implementation, it is the most efficient as it saves on context switching costs.

Following our decision several implementation issues had to be resolved. Among the most important are "safe" context swaps and preventing stack overflows. Because there is no intermediate stack when a context switch occurs, there must be always a ready thread to provide the stack for the switching routines. We have solved this problem by providing an "idle" thread, which is invisible to applications and never blocks. This way it is guaranteed that for every blocking thread there will be a ready thread to switch to. Since executing TCE kernel functions on application thread stacks can cause stack overflows, all stacks are enlarged by the amount needed for kernel executions. We put special attention to eliminating unnecessary calls and on-stack variables, in order to keep the extra stack space small.

Another important decision we have been faced is to ensure atomicity of execution for critical sections of the kernel code, in the preemptive mode. We have chosen to guard all TCE functions, instead of only specific implementation areas. The last is a better solution when responsiveness is concerned, but it would complicate the code significantly. A simple, working solution has been adopted that marks individually every thread which enters the TCE kernel as non-preemptive. When the thread returns from the kernel call, its flag is cleared. All signal handlers check the current thread's preemptive flag, and when it is set they will immediately terminate their actions. At these instances signals are lost, so to prevent it signal handlers leave their flags set. Handler flags are inspected in the "safe" areas of the TCE code, and if necessary the handlers can be activated from there.

4.3.1.1 Basic Thread Operations

Initially, all started processes contain only one thread of execution. By calling the TCE initialization function (tce_int), the C main function becomes the first TCE thread. The second thread created at that

instance is the hidden, “idle” thread, which by default has the lowest priority and is to be scheduled only when there are no other threads to run. New threads can be created any time by the tce_thrd_init library call. This call takes an address of a function which is going to become a new thread, and pointers to function parameters.

```
void func(TCE_Thrd ** pthrd)
{
   ...
   tce_thrd_detach(*pthrd); /* disjoint from parent */
   ...
}
....
void main(int argc, char *argv[])
{
    TCE_Thrd *child,*parent;
    ....
    tce_init();
    parent = tce_thrd_self(); /* pointer to itself */
    tce_thrd_stack(1024);
    ...
    child = tce_thrd_init(func,&parent);
    tce_thrd_priority(child,31);
    tce_thrd_start(child); /* start the child thread */
    ...
    tce_thrd_stop(child);  /* remove child from its queue */
    ...
    tce_thrd_start(child);
    ...
    tce_thrd_timeout(1024);/* wait 1024 clock ticks */
    ...
    tce_thrd_join(child); /* join child if it is still attached */
}
```

Figure 4: Basic operations on threads

The creating function returns a pointer to the newly created thread. Although the thread has been created, it is not made ready yet. It will be physically put to an appropriate priority queue only after a call to tce_thrd_start. The reason for it is to permit building a pool of ready threads, which can be activated later on. Taking a thread out of its scheduling queue is supported via the tce_thrd_stop. Programs can take advantage of these functions to implement their own, customized scheduling.

A priority of a new thread is the same as the parent’s priority. The tce_thrd_priority call is provided to change priorities when desired.

We decided to implement the parent-child relation for TCE threads. This relation prohibits a parent to exit before all its children terminate. There are two important reasons for this determination — a clean semantics induced by a tree of invocations and a preservation of valid references to the parent on-stack variables even when the parent has finished its execution To allow a parent thread to await its child termination the tce_thrd_join call has been provided. This call will block the calling thread when a child has not terminated yet. Otherwise, it returns immediately. Both a parent and a child are free to terminate their relation by calling tce_thrd_detach.

Timeouts are another helpful functionality concerning threads, that has been provided in TCE. Threads can request timeouts calling tce_thrd_timeout, submitting number of clock ticks (see next section) as a blocking interval. As a result of the timeout call, the requesting thread will be blocked until the specified number of ticks elapses. This capability supports building monitoring threads (“watchdogs”), which are required for implementing time dependent actions useful in multimedia, windowing or other responsive environments.

A portable, user-level implementation of multithreading cannot provide transparently growing and shrinking stacks. It is required from an application to declare a new stack size limit when one of its threads is going to use a stack larger than the 4KB default limit. The tce_thrd_stack call sets a new stack size, which will be the default for all newly created threads. The stack limit can be either risen or lowered. As we mentioned

it previously, stack size declared by an application is actually enlarged by an amount necessary to carry out execution of the kernel function.

4.3.1.2 Scheduling

The scheduling policy of our library is based on priorities. The priority range is between 0 and 31, where 31 is the highest priority. This is in contrast to the traditional approach in Operating Systems, where lower numbers represent higher priorities. However, in our environment priorities can be manipulated freely, so we have chosen higher numbers for higher priorities as more intuitive.

Each priority has its queue of ready threads, which can be empty (no threads of this priority ready to run). The scheduler will always choose the highest priority ready thread. If there is more than one thread in the highest priority queue, these threads are scheduled in the round robin fashion. If all application threads are blocked, meaning all priorities queues are empty, TCE starts its "idle" thread. That thread just waits for an external event (such as a message or an OS signal), and will relinquish control when any of the application threads become ready.

An essential question was whether the TCE scheduling should be preemptive or non-preemptive. The answer to which scheduling policy is better is application dependent. Non-preemptive execution is in most cases a more efficient way to carry out computation. It leaves the context switching costs under an application control, prevents from disrupting caches and saves on usually expensive signal (interrupt) handling. These benefits come at the expense of a limited responsiveness, lack of real-time guarantees, and a possible unfairness in the access to CPU(s) and other resources. In most cases, parallel applications do fine with a non-preemptive scheduling. However, multiuser servers as well as parallel solvers for irregular, dynamically adaptable problems, demand the preemptive execution. Trying to satisfy those contradictory needs, we decided to leave the decision about the enforced scheduling policy to an application.

The default scheduling mode in TCE is non-preemptive. A different thread can be started or resumed in its execution only when the thread which is currently running blocks itself or explicitly yields control (tce_thrd_yield). Also only in these instances the network is polled for new messages or for completions of outstanding sends.

```
...
int priority = 31;
float tick = .5  /* 500 us tick */
int slice = 10;  /* 10 ticks = .5*10 ms */
...
tce_preempt_on(); /* switch preemptiveness on */
...
tce_clock_tick(tick); /* setting the clock tick */
tce_clock_slice(priority,slice); /* slice for priority 31 */
...
tce_signals_off(); /* switch off signals */
....
tce_signals_on();
 ...
tce_preempt_off();/* switch preemptiveness off */
...
```

Figure 5: Setting the scheduling attributes

The preemptive scheduling scheme can be activated and deactivated any time by calling tce_preempt_on and tce_preempt_off. The preemptive mode will enforce context switching due to two possible conditions :

— the currently executed thread has exhausted its time slice and there are other ready threads with priorities not smaller than the current thread's priority

— the interrupt driven network manager has made a previously blocked thread ready, and that thread's priority is higher than the priority of the presently scheduled thread.

Time slices for priority queues as well as the frequency of clock ticks are under an application control.

Often, communication oriented applications want to obtain a maximum responsiveness when the sending and the receiving messages is concerned, but careless about their threads having a suitable share of CPU(s). Setting the clock tick to a non-positive value disables the clock, thus only communication will cause preemptions.

Preemptive scheduling can change the order of execution at any instance, so critical sections in a program should be guarded by semaphores. Another method is to temporary disable signals, do the critical work, and then enable them again. This method has to be used when dealing with non-reentrant system services (malloc for example) or to avoid interrupting system calls which are not restarted automatically. To minimize the use of this cumbersome solution the TCE library offers its own malloc and free calls — tce_malloc and tce_free. A program should always use them while in the preemptive mode, because they allocate and free memory safely and more efficiently. All TCE interface calls are guaranteed to be atomic.

4.3.1.3 Synchronization

In a multithreaded environment a synchronization mechanism is required to ensure a desired sequence of execution. Most light-weight thread environments offer mutexes with conditional variables. Counting semaphores are more powerful and flexible synchronization mechanism, but usually less efficient than mutexes. Because we have implemented semaphores entirely in the user space (with all signals disabled), their implementation overhead is negligible when compared to mutexes. On a shared memory, multiprocessor hardware locking has to be performed to secure mutual exclusion. This may render semaphores as to costly, hence following the Amoeba solution we intend to support both semaphores and mutexes.

```
TCE_Sem *sem,*wait;
...
void func1(int *p1)
{
    ...
    tce_sem_down(sem);       /* enter critical section */
    ...                      /* do some work */
    tce_sem_updown(sem,wait);/* leave critical section, and wait for parent */
}
void func2(int *p2)
{
    ...
   tce_sem_down(sem);
    ...
   tce_sem_updown(sem,wait);
}
void main(int argc, char *argv[])
{
    TCE_Thrd *thrd1,*thrd2;
    int prm1,prm2;
    ...
    wait = tce_sem_init(0); /* create a semaphore, set the counter to 0 */
    sem = tce_sem_init(1);
    thrd1 = tce_thrd_init(func1,&prm1);
    tce_thrd_start(thrd1);
    thrd2 = tce_thrd_init(func2,&prm2);
    tce_thrd_start(thrd2);
    ...
    tce_sem_up(wait);      /* let them proceed */
    tce_sem_up(wait);
}
```

Figure 6: Example of using TCE semaphores

Currently we have six functions which operate on semaphores:

- tce_sem_init — creates a semaphore
- tce_sem_destroy — destroys a semaphore
- tce_sem_set — sets a value for a semaphore counter
- tce_sem_up — increments the semaphore counter, possibly unblocking one of the blocked threads

- tce_sem_down — tries to decrement it, if the value is already 0 the calling thread will be blocked
- tce_sem_updown — operates atomically on two semaphores — one is pulled up and the other is pulled down. By using this function it is possible to obtain the same functionality provided by mutexes with conditional variables.

4.3.2 Communication

The TCE communication is carried out through abstract communication objects called channels. Another abstract communication type is port, which is used to establish communication channels. One channel connects exclusively two processes and it is up these processes to decide which threads will be writing or/and reading to/from that channel. We have based the communication on abstract objects to better shield application from various communication differences on diverse platforms. It is the TCE responsibility to assure an optimal communication, accommodating the most appropriate mechanisms available on a given system. Consequently, despite the fact that the actual implementation may use sockets, message passing primitives, remote references or Active Messages, applications can always employ channels for their communication.

Operations on channels have the message passing syntax and semantics. We have selected message passing model as the communication paradigm because of its efficiency and the implicit synchronization between a sender and a receiver. If other styles of interactions are desired, they can be always realized by operations on threads and channels.

4.3.2.1 Ports

Ports are essential ingredients of the TCE model. We have introduced them to decouple verification and authentication from regular operations on channels. By controlling the access to the receiving port, a process can avoid tampering its internal state by malicious processes. Ports permit creating more than one channel per connection, as well as specifying whether the channel is two-way, one-way or null. We took advantage of ports functionality in providing a location transparent location of registered services and building specialized channels for exceptions and tty interactions.

Ports can be treated as processes' points of contact. There are four standard ports which are automatically created for all processes:

— TCE_InPort — general receiving port, it declares that a channel receiving direction should be enabled .

— TCE_OutPort — general sending port, this port is to be used to create channels between an application and the TCE kernel with the unblocked sending direction.

— TCE_ExcPort — exception port, used to create channels on which exceptions, generated by the TCE kernel, are reported.

— TCE_NullPort — port used to close one or both directions on a channel.

In addition to standard ports, any number of regular sending ports can be opened. A regular sending port represents a sending right to a processes' receiving port. It can be treated as an image of a process' TCE_InPort.

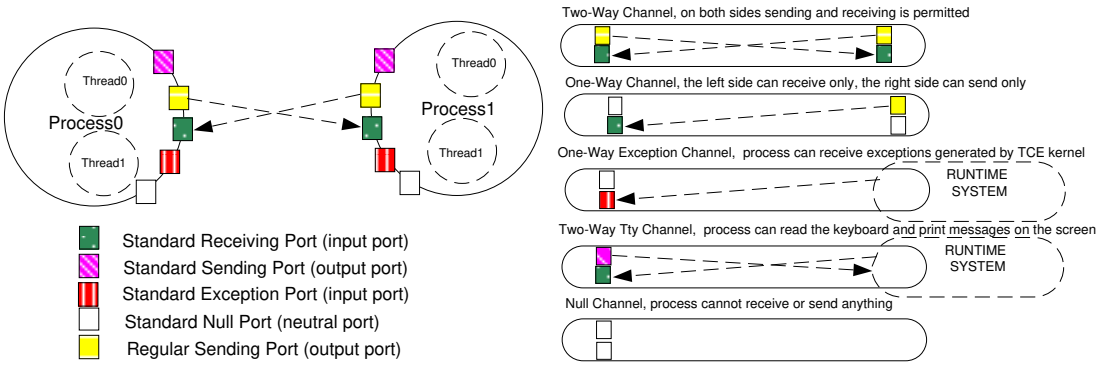


Figure 7: a) TCE Port Types b) TCE Channel Types

Regular sending ports are obtained as a result of a process creation or by receiving them in a message (as depicted in Figure 8). In the following example the SendingPorts array will be filled out by ports created as a result of starting NumOfNodes copies of the SlaveName program on Nodes nodes. By using these ports the parent process can now establish dedicated channels to each of its children by calling tce_chn_init. Through the channels it can pass the sending ports to all its children in order to let them create their own channels.

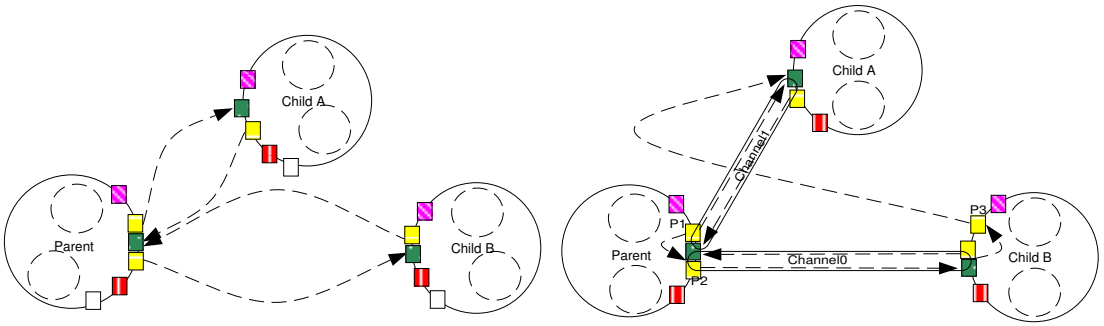
```
#define NumOfNodes 8
#define SlaveName "/usr/janusz/slave_program"

TCE_Port *SendingPorts[NumOfNodes - 1];
TCE_Port *ReceivingPorts[NumOfNodes - 1];

TCE_Chn *Channels[NumOfNodes -1];
int i;
...
n  = tce_exec(SendingPorts,Nodes,SlaveName,TCE_InPort,NumOfNodes - 1);
for(i = 0; i < n; i++)
  ReceivingPorts[i] = TCE_InPort; /* give every child access to parent's port */
tce_chn_init(Channels,ReceivingPorts,SendingPorts,n);
for(i = 0; i < n; i++)
    tce_chn_snd(Channels[i],SendingPorts,n,TCE_FMT_Port);
...
```

Figure 8: Example of creating regular sending ports

As the result of the spawning operation, a newly created child process always returns the sending right to its receiving port. The parent, on the other hand, may pass the standard null port as the argument to the new process, preventing this way the child from opening a talking connection (Figures 8, 9a).



1) Parent creates the Children Processes
2) As a result, Parent gets the Sending Ports (sending rights) to the Children's Standard Receiving Ports
3) Children get Sending Ports to the Parent's Standard Receiving Port

1) Parent sends the Port P1 through the Channel Channel2
2) Child B receives this Port, which is installed as P3
3) The P3 Port points at the Child A's Standard Receiving Port

Figure 9: Creating Regular Sending Ports a) through process creation b) by sending them in a message

There is yet another way to obtain sending rights to somebody's receiving port. This method is based on referring to the previously globally registered receiving ports by their literal names (see Figure 10A client can locate a service of interest knowing only a receiving port name. The TCE network of daemons keeps the global database of registered ports. Different ports registered under the same name are treated as a replicated receiving port. Thus concurrent requests referring to that name will most likely receive accesses to different ports (depending upon load balance and availability).). This functionality is provided as a support for location transparent, parallel services.

```
/* server side */
...
tce_port_register("Service0",TCE_InPort);
...
tce_port_remove("Service0");  /* remove the declared service */
...
/* client side */
...
TCE_Port *ServicePort;
...
ServicePort = tce_port_request("Service0");
...
```

Figure 10: Obtaining sending ports via the naming service

4.3.2.2 Channels

A channel represents a communication link between two peer sites. To establish a new channel two tce_chn_init requests have to posted, both with a matching pair of ports. The involved sites are depicted by appropriate sending ports and they can be two threads belonging to two TCE processes, the same process or a thread and the TCE runtime system. Because a channel is an abstract entity, it can be constructed across machine, partition or other physical boundaries, potentially connecting processes scattered freely in the Internet-wide domain. To our knowledge, among all communication libraries only PVM can offer a comparable communication transparency.

A channel is a dynamically created object, hence it is only natural to supply a symmetric operation that closes a channel and frees the memory associated with it. This is performed by the tce_chn_close call. Likewise the init function, closing a channel requires the two connected sites to issue matching requests. If a closed channel has any threads still blocked on it, this is treated as an application error. Such threads are made ready and appropriate exception is generated.

Despite their advantages, abstract channels inflict one annoying necessity — applications have to create and destroy them explicitly. In order to make these operations simpler, we supplemented tce_chn_init and tce_chn_close with capabilities to create and destroy multiple channels in one call (Figure 11). Their implementation guarantees that properly formulated requests will be carried out in a deadlock-free fashion.

```
...

TCE_Port *ReceivingPorts[8],*SendingPorts[8];
TCE_Chn *Chns[8];
int n = 8;
...

tce_chn_init(Chns,ReceivingPorts,SendingPorts,n);
...
tce_chn_close(Chns,n);
...
```

Figure 11: Creation and destruction of channels

By submitting appropriate port types it is possible to create different channel types. A regular, two way channel is created when the matching pairs contain the standard TCE_InPort receiving port and a sending port (Figure 12 a). If instead, other ports are given one of the special channels will be constructed. Correspondingly to ports, all regular and special channels can be passed in a message (Figure 12b). The side that sends a channel, however, looses its access to it as a channel can link only two peers.

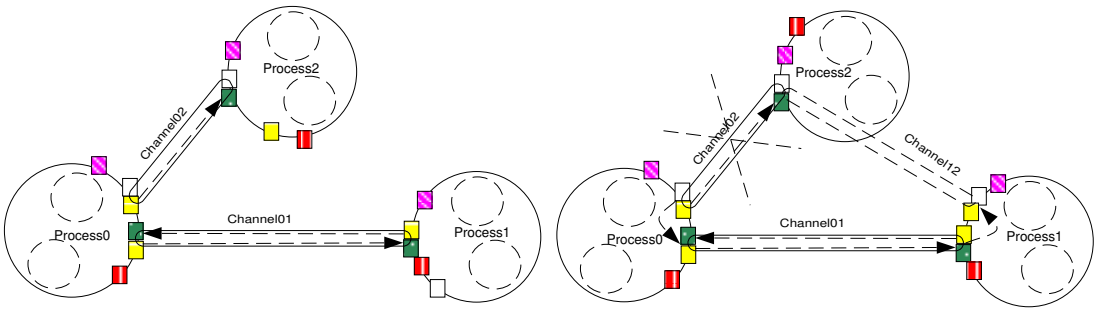


Figure 12: Channel creation a) by using ports b) by sending it in a message

There are only two basic communication calls which operate on a channel:

— tce_chn_snd — sends a message, a sending thread is blocked until the operation is completed.

— tce_chn_rcv — receives a message on the channel, the receiving thread is blocked when no matching message is available yet.

Both calls have blocking semantics at the application level. It means that a thread that issued a call which cannot be fulfilled immediately will be blocked and resumed later, when the operation has been completed. Obviously, our implementation translates all those communication requests into non-blocking system calls and makes scheduling decision in accordance with completion of those.

```
...
/* sends n integers on the first channel */

n = tce_chn_snd(Chns[0],buffer,len,TCE_FMT_Int);
...
/* receives n integers */
n = tce_chn_rcv(Chns[0],buffer,len,TCE_FMT_Int);
...
/* sends 8 sending ports */
n = tce_chn_snd(Chns[1],SendingPorts,TCE_FMT_Port,8);
/* receives 1 channel */
n = tce_chn_snd(Chns[1],&RChn,TCE_FMT_Chn,1);
...
```

Figure 13: Different sends and receives on a channel

Our environment tries to abstract physical resources, therefore a natural consequence is the automatic conversion between different data representations. As depicted in Figure 13, each communication call asks for a type of the buffer items. When a message has to be sent to a machine with a different byte order, the user buffer is XDR encoded. A complementary XDR decoding takes place at the receiving side, so a thread gets the buffer in the native data format. The user program can enforce no encoding at all by defining a data type as TCE_FMT_Byte. To make heterogenous communication more efficient we are referring to the XDR encoding only for floating point types. All other types are converted internally by TCE at a receiving side.

Besides heterogenous communication, supplying data types permits us to pretend that passing ports and channels can be done the same way as ordinary data type transfers. Actually, sending and receiving communication objects has to be implemented differently, but applications are not able to see the difference. As the result, a polymorphic and expressive characteristic of the communication objects has been obtained.

Sending communication objects as ordinary data was the major reason why tce_chn_rcv and tce_chn_snd accept the length of a buffer in data items instead of bytes. The last would be meaningless while transferring ports and channels. The other cause is our conviction that specifying number of items is simply more intuitive because it resembles the way language control loops are written.

A call complementing the channel specific operations is tce_chn_set. It imposes different communication characteristics on a channel. Switching to a new channel mode needs both sides connected by that channel to issue identical operations. Otherwise, involved threads are blocked, waiting for an appropriate type of the setting. Likewise all other functions which operate on multiple ports and channels. this call is guaranteed to change the characteristics of multiple channels in a deadlock free way, as long as the requests are formulated properly.

```
...
/* imposes synchronous message passing on n channels */
res = tce_chn_set(Chns,TCE_CHN_Sync,1,n);
...
/* switches back to the default communication mode */
res = tce_chn_set(Chns,TCE_CHN_Sync,0,n);
...
/* requests the force mode */
res = tce_chn_set(Chns,TCE_CHN_Force,1,n);
...
```

Figure 14: Imposing different communication modes

The tce_chn_set function allows to tailor the message passing protocol on chosen channels. TCE supports three message passing modes:

— regular mode — TCE_CHN_Reg, which buffers a message on the receiving side, when no thread waits for it.

— synchronous mode — TCE_CHN_Sync, which postpones a send, until a matching receive is posted. This assures that the message does not have to be buffered at the receiver's side. On systems which do not support the synchronous mode directly, this functionality comes at the expense of one extra message sent by the receiving port.

— force mode — TCE_CHN_Force which discards the received message when nobody waits for it. This mode is mainly introduced to utilize a fast, handshake-free communication provided by some architectures such as iPCS/860 [21]. For architectures which don't have this option the TCE implementation can either keep discarding messages that nobody waits for (stream oriented transport protocols) or quietly turn the force mode into the regular mode (datagram oriented transport protocols).

We have decided to provide these three modes in message passing since different parallel programs show different needs for buffering and handshaking. For small messages, buffering with minimal handshake can result in a better performance, as buffering has smoothing effects for loosely synchronized processes. On the other hand, large data transfers will find it beneficial to refer to the synchronous message passing, because the costs of allocating large memory chunks as well as copying many items can largely reduce the expected efficiency. Finally, very well synchronized programs or applications which can tolerate a loss of messages, can take advantage of the force mode and this way boost the performance.

```
...
/* requires two acks for each message sent - from a node and a thread */
res = tce_chn_set(Chns,TCE_CHN_AckNode | TCE_CHN_AckThrd,1,n);
...
/* sets tag 13 on n channels */
res = tce_chn_set(&Chns,TCE_CHN_Tag,13,n);
...
```

Figure 15: Additional channel operations

The same TCE call can be used to request acknowledgments or to change a channel tag (Figure 15). Acknowledgments can be generated either when a message actually reaches a destination node or when it is placed in the thread's receiving buffer — TCE_CHN_AckNode, TCE_CHN_AckThrd. A sending thread is left blocked until the expected acknowledgment arrives. Their main purpose is to help in debugging and,

when coupled with timeouts, to provide a support for building customized, reliable protocols over non-reliable transport protocols.

Different tags can be declared for channels in order to precisely address a message that a thread expects to receive or carry across. Tags also make it possible to differentiate between multiple actions, concurrently active on given channels.

4.3.2.3 Special Channels

As we mentioned earlier, by using appropriate port types it is possible to create different types of channels (Figure 7b).

One-way or null channels can be built using the null port abstraction. To build a one-way channel one side has to submit the standard receiving port and the null port, the other side the sending port and the null port. If instead, both submitted ports are null a null channel will be returned. All operations which refer to blocked (null) directions are implemented as empty functions, always returning 0 items sent or received.

The one-directional channels have been provided to lay foundations for the client-server interactions. At the beginning, only a client knows the server's receiving port that was registered under a given name. A server cannot know a client's receiving port, because this information is available only at the moment when a client requests that service. When a one-way channel has been established, the participating client can send the access right to its receiving port through that channel. Thus, the client and the server can build a new channel, this time two-directional.

We have included the null channel type to help parallel applications which have to deal with inter-neighbor communications under boundary conditions. Instead of constantly checking whether a subsequent communication link is valid the non existing links can be represented as null channels and the code will be considerably simplified.

In addition to one-way and null channels, an application can request two additional special channel types — a tty channel and an exception channel.

To build the first one, a program has to provide the standard sending and receiving ports — TCE_OutPort, TCE_InPort, to tce_chn_init. A presence of a tty channel will instruct the TCE kernel to treat any tty input as a message being delivered to that channel. All writes to a tty channel are considered as ascii outputs directed to the screen.

```
TCE_Chn *TtyChn;
char cp[80];
char *sayHi = "Hi, tty ...";
...

tce_chn_init(&TtyChn,&TCE_InPort,&TCE_OutPort,1);
...
tce_chn_rcv(TtyChn,cp,80,TCE_FMT_Char);
if(!strcmp(cp,"Hello"))
   tce_chn_snd(TtyChn,sayHi,strlen(sayHi) + 1,TCE_FMT_Char);
...
```

Figure 16: Example of dealing with a tty channel

The exception channel can be initialized by providing tce_chn_init with the TCE_ExcPort port and the null port (Figure 17). It means that the exception channel can only be one directional — a process is allowed to receive exceptions but cannot generate them. All non fatal runtime errors discovered by the TCE kernel are turned into exception messages and delivered to an exception channel. If there is a blocked thread on that channel and its message tag matches the error number, then the appropriate message is delivered to that thread and the application can proceed. Otherwise, the application is terminated. A thread that has received an exception automatically gets the highest priority and will continue to run until it terminates or blocks itself. It is up to a user program to decide whether the exceptions of interest should be handled by multiple threads, a single thread, via a single or multiple exception channel(s). The exception message contains a triplet —

{Node,SendingPort,Thread} and a string describing the error. Using this information makes it is possible to decide whether the process should terminate or not. In the later case, all non-valid ports and channels as well as faulty threads can be identified and the problematic situation fixed.

```
TCE_Chn *ExcChn;
TCE_ExcMsg *ExcMsg;
...
tce_chn_init(&ExcChn,&TCE_ExcPort,&TCE_Null,1);
...
/* report all exceptions on this channel */
tce_chn_set(&ExcChn,TCE_CHN_Tag,-1,1);
tce_chn_rcv(ExcChn,&ExcMsg,0,TCE_FMT_ExcMsg);
printf("\n Error: %s ",ExcMsg->String);
tce_exit(1);
...
```

Figure 17: Exception channel

We provided the tty and exception channel capabilities to obtain a uniform treatment for message passing, tty interactions and error checking. Those channels make also writing efficient applications an easier task, as the tty requests block only one thread instead of the whole process and exception handling frees a program from checking every TCE call for an error. We expect the last to be much appreciated by many programmers, who now are forced to look for an error in each system or library call, only to terminate their programs when such has been found.

4.3.2.4 Channel Sets

Channels permit only peer-to-peer communication. In order to allow a collective interaction without specifying all participants individually, an object called channel set is provided. On channel sets collective communication calls are defined. Collective calls are very relevant for data parallel programs, where a collection of precesses operate cooperatively on uniformly distributed data sets. Their main advantages are simplified parallel codes and performance gains, as their implementation optimally utilizes multiple communication links.

A TCE channel set is created by the tce_chnset_init function which has to be invoked with an array of channels as its parameter. Over that array of channels the channel set is going to be spanned. All sides having access to those channels must participate in this creation process, submitting identical arrays of channels. As the result, they receive a pointer to the channel set object.

```
TCE_ChnSet *ChnSet;
TCE_Chn *Chns[8];
int n = 8;
...
/* creating a set of 8 channels */
n = tce_chnset_init(&ChnSet,Chns,n);
...
/* setting a channel set tag */
tce_chnset_set(ChnSet,TCE_CHNSET_Tag,14);
....
/* removing all channels from the set = destroying the set */
tce_chnset_close(ChnSet,&Chn[0],n);
...
```

Figure 18: Basic calls for channel sets

To remove a given channel from a set or to destroy the set completely, all participating sites have to issue the tce_chnset_close call simultaneously. A channel cannot be added dynamically to an existing set. This effect may be achieved, however, by first closing a set and then creating a new one with some new channels added. It is possible to implement the group adjustment in a dynamic way, using multiple threads. Therefore, the initial static group semantics can be easily expanded to a dynamic one by an application itself.

Unlike channels, tce_chnset_set does not allow to tailor any of the message passing details. For all collective communications those details are taken care of internally by the TCE implementation, which needs this kind of flexibility to implement them efficiently for different parallel platforms.

On a contrary to collective communications presented by other parallel libraries [17,27,32] our collective calls don't block applications process, but only involved threads. This feature has two important repercussions. Firstly, our implementation cannot be based on collective operations supplied by the HPC vendors as they don't support non-blocking counterparts. This is rather unfortunate, since some vendor implementations are efficiently supported by hardware [34]. Secondly, many collective operations can be concurrently active on a given channel set. Obviously, this is a desirable situation as we strive to provide as much concurrency as possible. However, this degree of concurrency can produce flow control problems. In order to prevent them, applications should synchronize between consecutive collective calls (tce_chnset_sync) if the amount of memory space necessary for their execution is substantial. When the same collective calls are concurrently active on a channel set and they use the same data format, a program should use different channel tags for them. Otherwise, the TCE runtime will have no means to differentiate between them.

```
#define len 1024
#define circular 1
...
extern void comp(int *, int *, int );

TCE_ChnSet *ChnSet;
int buffer[len],inbuf[len],outbuf[len];
int src,step = 1;
...
src = tce_chnset_rank(ChnSet);
tce_chnset_sync(ChnSet);
tce_chnset_bcast(ChnSet,src,buffer,len,TCE_FMT_Int);
tce_chnset_combine(ChnSet,comp,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_concat(ChnSet,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_gather(ChnSet,src,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_prefix(ChnSet,comp,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_reduce(ChnSet,src,comp,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_scatter(ChnSet,src,inbuf,outbuf,len,TCE_FMT_Int);
tce_chnset_shift(ChnSet,circular,step,inbuf,outbuf,len,TCE_FMT_Int);
...
```

Figure 19: Collective communications

Because channel sets are spanned over a vector of channels, it is possible that one or more channels link threads within the same process. Therefore a rank within a channel set does not have to correspond to relative ranks of processes which are connected by a channel set. The rank only represents a position in the ordering of the channel set "ends" for a given channel set.

The following collective calls are currently supported in TCE:

- tce_chnset_sync — performs a barrier synchronization. It returns successfully only when on all objects representing the same channel set matching operation has been posted.
- tce_chnset_bcast — broadcasts a message from a one channel set participant to all other.
- tce_chnset_combine — applies an associative reduction operation, defined by the user supplied function, and broadcasts the result to all threads which participate in this operation on a given channel set.
- tce_chnset_concat — concatenates all presented buffers into one vector which is then broadcasted to all threads participating in this operation on a particular channel set.
- tce_chnset_gather — gathers distinct messages into one buffer, which is than delivered to a single thread's buffer.
- tce_chnset_prefix — applies parallel prefix with respect to an associative reduction operation, defined by the user supplied function, across all participating threads on a channel set, and places the corresponding results in threads' buffers.

- tce_chnset_reduce — applies an associative reduction operation, defined by the user supplied function, and places the result in the thread's buffer on specified channel set end.
- tce_chnset_scatter — performs a reverse operation to tce_chnset_gather
- tce_chnset_shift — shifts participating threads' buffers left or right by a specified number of steps. Depending upon a submitted flag the shift is performed with or without wraparound.

5 Benchmark of the TCE Implementation

In this section we present results of benchmarking different aspects of the TCE library. The measurements concerned the implementation efficiency of multithreading and communication. We also tested performance of a sample parallel application which uses TCE.

A cluster of 8 DEC 3000 Model 400 AXP workstations, running OSF/1 V3.0 Operating System has been used for the measurements. The machines are connected by the GIGAswitch/FDDI switch, with the crossbar topology of links, providing 100Mb/s of bandwidth per link. Each workstation is equipped with the DECchip 21064, clocked at 133MHz. The Alpha processor has two on chip caches — for data and instructions — 8 KB each. The on board cache has 256KB and the main memory has 128MB. The IO system is equipped with the TURBOChannel architecture delivering 90MB/s of maximum throughput. The inter-machine communication is carried out by the FDDIcontroller/TURBOchannel adapter (DEFTA).

All benchmark results have been obtained while having an exclusive access to the cluster.

5.1 Multithreading Operations

The performance of basic thread operations is presented in Table 1. For all measurements the standard thread stack of 4KB has been used. Currently the package supports only passing a single argument to a new thread. Thus, measurements for variable number of arguments are not present here. Similarly, only the context switches involving both integer and floating point registers have been timed, since it is the only mode supported by TCE.

The most crucial results concern the context switching time and the creation and destruction time. The context switching operation, represented by tce_thrd_yield, takes 2.0us. To create and destroy a thread which is present in the TCE internal cache of threads requires 4.1us. The tce_thrd_yield call not only forces a context switch but also selects the highest priority thread which is ready to be scheduled. The creation and destruction is much more costly when there is a miss in the internal cache. In this case, both malloc and free system calls are used, which rises the overall time up to 45us. It takes .40us to move a thread from its blocked state to the ready state and the transition in the opposite direction needs .54us. These times do not include context swaps, but represent only costs of all necessary actions to put a thread into its scheduling queue and to remove it from there, selecting at the same time a new thread. Atomic actions which increment and decrement a semaphore require .31us and .28us respectively. These operations don't make any thread ready or blocked. The time needed to increment a semaphore which results in making a thread ready can be calculated by adding the time needed for the transition from the blocked to ready state (approximately .78us in total). A similar calculation can be performed for the blocking tce_sem_down call.

OPERATION	TIME [us]
yielding control to another thread - tce_thrd_yield()	2.0
thread creation + destruction - thread in the internal cache	4.1
thread creation + destruction - thread not in the cache	45.0
transition: blocked state -> ready state	.40
transition: ready state -> blocked state	.54
tce_sem_up(sem) - no thread blocked on the semaphore	.31
tce_sem_down(sem) - no blocking	.28

Table: 1 Performance of the TCE multithreading

5.2 Communication Operations

The TCE implementation on the Alpha workstations uses the OSF/1 socket interface to the TCP/IP stack. The size of the allocated TCP input and output buffers is set to 100KB. All sockets are defined as non-blocking and the select system call is used to determine on which socket writing or reading is possible. For sending a buffer, the TCE network subsystem uses the writev system call and for receiving the readv. Both calls utilize two entries in the iovec vector of buffer pointers.

As Table 2 shows, empty writev and readv system calls take 39.2us and 16.7 respectively. In addition to this overhead, every writev/readv system call has to be followed/proceeded by two memory-to-memory copies and one checksum calculation [4]. One copying process involves transferring bytes between the mbufs kernel buffers and the user buffers through the memory bus, the other one between mbufs and the FDDI adapter memory via the system IO bus. An experimental value of .024us per byte depicts costs of those operations. The measurements suggest that the overhead introduced by TCE should be negligible, for example an emulation of a single, non-blocking receive using TCE threads and semaphores has an upper bound of 12.3us, which is less than 11% of the duration of readv receiving 4KB packet and only .96% of latency.

OPERATION	TIME [us]
empty writev system call	39.2
empty readv system call	16.7
overhread introduced by tce_chn_snd	1.8
overhread introduced by tce_chn_rcv	1.6
round trip latency for TCP/IP using writev/readv	1280

Table 2: Execution times for OSF/1 and TCE communication calls

To evaluate the communication efficiency of TCE, we benchmarked point-to-point and collective communications for TCE and PVM – version 3.3.6. In both benchmarks TCE programs contained 2 threads — one primary thread and one idle thread. The idle thread was only active when the primary thread was blocked, waiting for a completion of read/write operations. In the point-to-point benchmark, the PVM program utilized the most efficient communication primitives pvm_psend and pvm_precv [17]. These primitives don't require an extra copying between the application buffers and PVM buffers. Message passing takes place via direct TCP connections, bypassing the pvmd daemons.

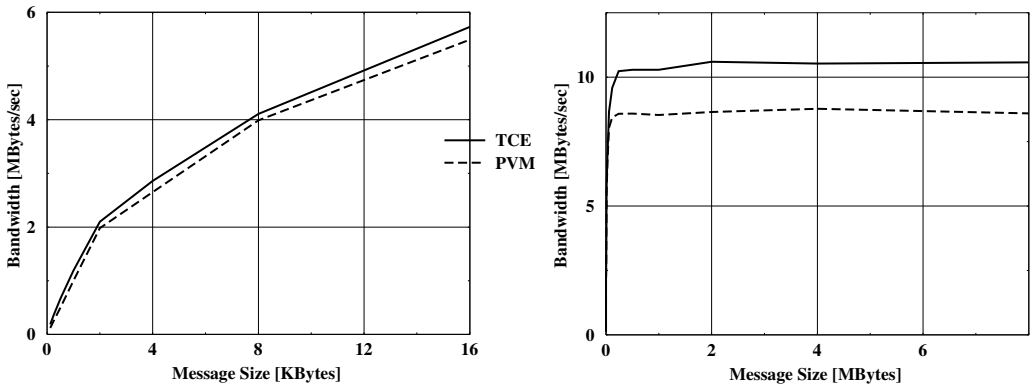


Figure 20: Point-to-point communication in TCE and PVM for small and large message sizes

Figure 20 presents the bandwidth obtainable through TCE and PVM point-to-point communication calls. It was measured as a round trip time, between two nodes, for different message sizes. Collective communications in Table 3 have been timed in a loop, each call has been followed by a barrier (barrier costs have been already subtracted). For collective calls all 8 alpha nodes have been involved. Analyzing the results, we can see that TCE performs consistently better for point-to-point communication across all message sizes. The dramatic difference in the case of collective communications is caused by the fact that PVM has to use daemons to provide the required group functionality.

\[ms]	barrier	bcast	gather	scatter	reduce (sum)	concat	combine (sum)	prefix (sum)	cshift (step 1)	shift (step 1)
TCE	2.8	3.5	5.2	3.8	5.6	24.7	9.5	9.3	5.5	4.8
PVM	15.4	30.1	44.7	75.2	45.4	N/A	N/A	N/A	N/A	N/A

Table 3: Performance of TCE and PVM collective communication calls (all calls except barrier operate on 1KB vector of integers).

5.3 Benchmarking a Parallel Application

It is obvious that TCE environment should perform better than non-threaded communication systems for irregular, asynchronous applications or applications which mix data and functional parallelism. The question remains, however, whether TCE can improve performance for regular, data parallel problems. The answer is important, since as research shows [15], 90% of problems applicable to parallel execution can be solved using synchronous or loosely synchronous, data parallel techniques. To investigate it, we have chosen the matrix multiplication problem, and implemented it using single and multiple threads of execution. The multithreaded implementation will be compared to a single threaded program that uses blocking communication calls. One could argue that an application with non-blocking communications should be used instead for that comparison. But in the case of user-level multithreading, programs which use non-blocking calls should always perform better than multithreaded counterparts, assuming that such programs can be constructed optimally. Our goal here is to investigate if the additional concurrency provided by threads can outweigh the overhead introduced by the multithreading implementation. The results of the previously presented communication benchmark justify the usage of a single threaded TCE program as the base for the comparison.

5.3.1 Matrix Multiplier

For the sake of the benchmark, it is assumed that we deal with square matrixes A, B, C. Rows of matrix A and columns of matrix B are evenly distributed across computing nodes. Each process has to compute the product of its local rows and columns and the product of its local rows and all remote columns

storing the results in its portion of the C matrix. There are other, more efficient algorithms in the terms of computation/communication ratio (for example Fox's algorithm [14]), but we have chosen this one due to its simplicity.

In the implementation of a "standard" matrix multiplier, all nodes multiply their local rows by columns in identical steps. In each iteration of the outermost loop, the location of an appropriate columns is calculated. If the desired column is local, the node broadcasts the column to all nodes. Otherwise, it waits for that column to be broadcasted from the node that owns it (Figure 21).

```
#define NN 8   /* number of nodes */
#define M 512  /* matrix size */
double A[M/NN][M];
double B[M/NN][M];
double C[M/NN][M];
double V0[M];
int rank;
TCE_ChnSet *ChnSet;
...
void
do_work()
{
     int i,j,k,l;
     int nnum;       /* node number */
     int cpn = M/NN; /* #columns per node */
     double *Vt;
     for(l = 0; l < M; l++)
     {
       nnum = l/cpn; /* calculates owner node */
       j =  l%cpn;   /* local column within */
       if(nnum == rank)
       Vt = &B[j][0];
     else
       Vt = V0;
     tce_chnset_bcast(ChnSet,nnum,Vt,M,TCE_FMT_Double);
      for(k = 0; k < M/NN;k++)
       for(i = 0; i < M; i++)
         C[k][l] = C[k][l] + A[k][i]*Vt[i];
     }
}
...
```

Figure 21: Matrix Multiplication using a single thread of execution

A fundamental inefficiency of that approach lays in a lack of overlapping between computation and communication. Given that nodes remain synchronized all the time, all receiving nodes have to wait for the length of time needed to transfer the whole column to them. Sending and receiving multiple columns in one operation will reduce the startup costs, but requires some extra storage.

While improving the implementation of the matrix multiplier, we tried to exploit multiple threads of execution to hide communication costs. Since multiplying local rows and columns is independent from computing a product of local rows and remote columns, designated independent threads carry out these operations. A thread doing calculations on local arrays has lower priority, thus, it can be regarded as a background thread. It will run only when there is no other work to do.

To exploit more concurrency, dedicated sending threads have been created. There is one sending thread per channel and each channel has the synchronous mode imposed on it. A sending thread is then blocked on its channel, awaiting for a request to send a subsequent column. When such a request has been received, the column is sent and the thread will proceed to sending the next column, if there are any left. This scheme eliminates any need for internal buffering.

```
void
compute_local()
{
     int i,j,k,t = 0;
     int start = rank*M/NN;
     int end = start + M/NN;
     for(j = start; j < end; j++,t++)
    for(k = 0; k < M/NN; k++)
    {
       for(i = 0; i < M; i++)
           C[k][j] = C[k][j] + A[k][i]*B[t][i];
       tce_thrd_yield();
    }
     tce_sem_up(End);
}
void
compute_remote()
{
     int i,j,k,l,cpn = M/NN;
     int start = (rank + 1)*cpn;
     int end = start + M - cpn;
     O_Thread *Thrd;
     RcvStart = tce_sm_init(1);
     RcvEnd = tce_sm_init(0);
     Thrd = tce_thrd_init(receive,(void *)0);
     tce_thrd_priority(Thrd,2);/* higher priority */
     tce_thrd_start(Thrd);     /* start the helping thread */
     tce_thrd_yield();
     for(l = start; l < end; l++)
     {
       j = l%M;
       tce_sem_down(RcvEnd);  /* request the current column */
       tce_sem_up(RcvStart);  /* let receive the next column */
       tce_thrd_yield();
       for(k = 0; k < M/NN;k++)
         for(i = 0; i < M; i++)
           C[k][j] = C[k][j] + A[k][i]*Vt[i];
     }
     tce_thrd_join(Thrd);
     tce_sem_up(End);
}
```

Figure 22: Threads performing computation using local and remote columns

```
void
send(TCE_Chn **chnp)
{
     int i;
     for(i = 0; i < M/NN; i++)
       tce_chn_snd(*chnp,&B[i][0],M,TCE_FMT_Double);
     tce_sem_up(End);
}
```

Figure 23: Sending threads

Finally, the receiving process has been enhanced by prefetching a column needed in the next step of computation. That prefetching is implemented as a "helping" thread, which blocks itself while waiting for a column, and then synchronizes with the thread that computes the local rows, remote columns product.

The core part of the discussed implementation is presented in Figures 22, 23, 24 and 25. The do_work function is a starting thread, which creates the two threads for computing; compute_local and compute_remote, and as many as the number of nodes minus 1 threads for sending columns. The compute_remote thread spawns a "helper" thread called receive which prefetches remote columns. The shown code will be executed in the non-preemptive mode. To get the preemptive execution, the comments surrounding tce_preempt_on should be removed (Figure 22). In the preemptive mode all yielding operations are redundant and can be taken away. The complete program is shown in Appendix B.

```
void
receive()
{
     int nnum,j,l,cpn = M/NN;
     int start = (rank + 1)*cpn;
     int end = start + M - cpn;
     double *Ptr;
     for(l = start; l < end; l++)
     {
       tce_sem_down(RcvStart); /* wait until previous column used */
       j = l%M;
       nnum = j/cpn;
       Ptr = &V0[l%2][0];     /* take a free vector */
       tce_chn_rcv(Chns[nnum],Ptr,M,TCE_FMT_Double);
       Vt = Ptr;
       tce_sem_up(RcvEnd);     /* notify that column received */
     }
}
```

Figure 24: Receiving thread which prefetches a remote column needed in the next computational step

```
void
do_work()
{
     TCE_Thrd *Thrd;
     int i;
     tce_chn_set(Chns,TCE_CHN_Sync,1,NN);
     Thrd = tce_thrd_init(compute_local,(void *)0);
     tce_thrd_start(Thrd); /* gets the same priority 0 */

     Thrd = tce_thrd_init(compute_remote,(void *)0);
     tce_thrd_priority(Thrd,1); /* gets higher priority */
     tce_thrd_start(Thrd);
     for(i = 0; i < NN; i++)
       if(i != rank)
       {
       Thrd = tce_thrd_init(send,&Chns[i]);
       tce_thrd_priority(Thrd,3); /* highest priority */
       tce_thrd_start(Thrd);
       }
     /* tce_preempt_on() */
     End = tce_sem_init(-NN);
     tce_sem_down(End);    /* wait for all started threads */
}
```

Figure 25: Starting threads for the multithreaded matrix multiplication

In order to assure the desired sequence of actions, two important TCE functions come in handy. A dynamic specification of the message passing protocol for channels allowed to specify channels as a request-reply link, where messages are only send when they are needed (TCE_CHN_Sync). Secondly, thread priorities enable a precise specification what to do if two or more conditions are simultaneously satisfied, for example the thread that uses remote columns for the calculations has a precedence over the thread which uses local rows and columns. Both these threads, however, will yield control when any of the "communicating" threads becomes ready.

Interestingly enough, the multithreaded algorithm for matrix multiplication very well illustrates how threads and channels can be used to implement the remote reference model. Unlike the remote reference method, in the case of threads and channels there is no need for an additional synchronization. A sender and a receiver are implicitly synchronized by the message passing semantics. Nevertheless, this benefit comes at the expense of additional context switching costs.

The results of comparison between the standard and the multithreaded matrix multiplier, presented in Figure 26, show that indeed the multithreaded implementation improves performance from 70% for small matrix sizes to 10% for large ones. The improvement gets smaller for larger matrixes as the computation part

dominates the communication. The preemptive execution still gives a noticeable gain, however, significantly smaller than the non-preemptive counterpart. In the case of preemptive multithreading, the improvement goes down more rapidly for large matrixes since the preemptive communication spoils cache localities more often. Cache locality for large sizes is essential for good performance as the communication costs become less important. Both curves show two valleys — they both correspond to the increase in cache misses caused by multithreading. The first, very sharp minimum is inflicted by the size of the primary data cache. The single threaded program is able to fit the needed matrix elements into it, the multithreaded program cannot, because it has two computational threads which have different access patterns to the A and C elements. Similar situation happens for the secondary cache, inducing the second local minimum.

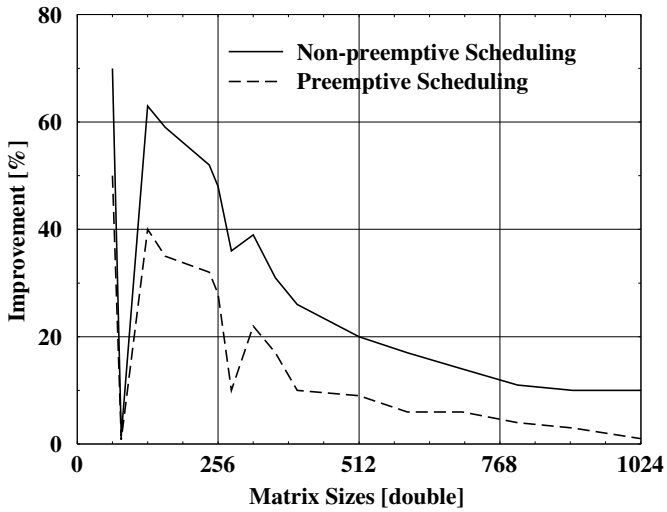


Figure 26: Matrix Multiplication — improvement using multithreading

The multithreaded algorithm may not scale up well, since each channel is being served by an independent thread. For large processor configuration the combined stack sizes of all sending threads may easily exceed acceptable limits. We experimented with a more efficient communication scheme, which uses the shift collective call to create a pipeline carrying a stream of columns. In a multithreading implementation the shift call has been performed asynchronously by an additional thread. The comparison between the standard and multithreaded pipelined algorithms has shown almost identical performance gains as the presented implementation.

Since the TCE multithreaded implementation proved to be helpful in improving the performance of one regular data parallel application, we can expect similar benefits for any other application which can effectively overlap communication with computation. This claim should come out true for all distributed memory systems which have the thread context switch time significantly smaller than the communication startup/latency costs.

6 Conclusion

This paper discusses the motivation, design choices and implementation of a new multithreaded communication environment called TCE. The environment is constructed as a compiled library, portable between different parallel and distributed systems. It provides a coherent support for data and functional parallelism via threading capabilities and an efficient interface to point-to-point and collective communications. The communication subsystem is based on abstract objects called ports, channels and channel sets. Operations on them are closely integrated with the thread scheduling. The TCE multithreading utilizes the user-level threads, and permits the application to choose appropriate scheduling attributes.

The results of our benchmark show that the TCE implementation is efficient both in terms of communication and multithreading. The analysis of the matrix multiplication application, coded in TCE, suggests that

our environment can be successfully applied in building efficient parallel solvers for both regular, synchronous and irregular/asynchronous problems.

Currently, the TCE library has been ported to the sun, mips and alpha architectures running SunOS 4.1.1, Ultrix 4.2 and OSF/1 3.0 respectively. Porting efforts to SP1/SP2 systems are underway.

7 References

[1] M. Acceta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, **"Mach: A New Kernel Foundations for Unix Development"**, Proc. Summer 1986 USENIX Conf, pages 93-112, 1986.

[2] R. Butler, E. Lusk, **"User's Guide to the p4 Programming System"** Argone National Lab, 1992.

[3] K.M. Chandy, C.Kesselman, **"CC++: A Declarative Concurrent Object Oriented Programming Notation"**, Technical Report, California Institute of Technology, March 12, 93.

[4] C. Chang, R. Flower, J. Forecast, H. Gray, W. R. Hawe, K.K. Ramakrishnan, A.P. Nadkarni, U.M. Shikarpur, K.M. Wilde, **"High-performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP"**, Digital Technical Journal Vol. 5 No. 1, 1993.

[5] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, and J. Wawrzynek, **"Fine-grain Parallelism with Minimal Hardware Support: A Compiler-controlled Threaded Abstract Machine"**, 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[6] H. Custer, **"Inside Windows NT"**, Microsoft Press, 1993.

[7] T.A. Dutton, D.E. Hugh, H.R. Kurth, J.J Reisert, and R.L. Stewart, **"The Design of the DEC 300 AXP Systems, Two High-performance Workstations"**, Digital Technical Journal Vol. 4 No. 4, 1992.

[8] T.von Eicken, D.E. Culler, S.C. Goldstain, and K.E. Schauser. **"Active messages: a mechanism for integrated communication and computation"**, In Proceedings of 19th International Symposium on Computer Architecture, pages 256-266, May 1992.

[9] M.J. Feeley, J.S. Chase, and E.D. Lazowska, **"User-Level Threads and Interprocess Communication"**, Technical Report 93–02–03, CSE University of Washington, Seatle, WA, 1993.

[10] E.W. Felten, **"Protocol Compilation: High-Performance Communication for Parallel Programs"**, Doctoral Thesis, University of Washington, 1993.

[11] E.W. Felten and D.McNamee **"Improving the Performance of Message-Passing Applications by Multithreading"** , Scalable High Performance Computing Conference, pages: 84–89, April 1992.

[12] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, **"Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems"**, Technical Report, Argonne National Labs, 1993.

[13] I. Foster, R. Olson, and S. Tuecke, **"Programming in Fortran M"**, Technical Report ANL-93/26, Argonne National Laboratory, 1993.

[14] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, and D.W. Walker. **"Solving Problems on Concurrent Processors"**, pages: 167–185, Prentice Hall, 1988.

[15] G.C. Fox, R.D. Williams, P.C. Messina, **"Parallel Computing Works!"**, pages: 603–645, Morgan Kaufmann Pusblishers, Inc, 1994.

[16] D. Gannon and J.K.Lee, **"Object Oriented Parallelism: pC++ Ideas and Experiments"**, Proc. of Japan Society for Parallel Processing, pages: 13-23., 1991.

[17] A. Geist, A. Beguelin, J. Dongara, W.Jiang, R. Manchek, V. Sunderam, **"PVM 3 User's Guide and Reference Manual"**, May 1994.

[18] M. Gien, **"Micro-kernel Architecture Key to Modern Operating Systems Design"**, Unix Review, Vol. 8, No. 11, November 1990.

[19] D. Grunwald, **"A User's Guide to AWESIME: An Object-oriented Parallel Programming and Simulation System"**, Technical Report CU-CS-552–91, University of Colorado at Boulder, November 1991.

[20] M. Haines, D. Cronk, and P. Mehrotra, **"On the Design of Chant: A Talking Thread Package"**, ICASE Report 94–26, NASA Langley Research Center, Hampton, VA, April 1994.

[21] Intel Corporation, **"iPSC/860 System User's Guide"**, 1992.

[22] L.V. Kale, S. Krishnan, **"CHARM++: A Portable Concurrent Object Oriented System Based On C++"**, Teachnical Report, University of Illinois, Urbana-Champaign.

[23] D. Keppel, **"Tools and Techniques for Building Fast Portable Threads Packages",** Technical Report UWCSE 93–05–06, University of Washington, May 1993.

[24] T.J. LeBlanc, E.P. Markatos, **"Shared Memory vs. Message Passing in Shared-Memory Multiprocessors"**, CS University of Rochester, Rochester, NY, April 1992.

[25] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, **" The Design and Implementation of the 4.3BSD UNIX Operating System"**, Addison-Wesley Publishing Company Inc., May 1989.

[26] B.D. Marsh, T.J. LeBlanc, M.L. Scott, E.P. Markatos, **"First-Class User-Level Threads"**, Proc. of the Thirteenth SCSP, pages: 110-121, Pacific Grove, CA, pages: 14-16 October 1991.

[27] Message Passing Interface Forum, **"MPI: A Message-Passing Interface Standard "**, May 1994 November 1993.

[28] F. Mueller, **"A Library Implementation of POSIX Threads under UNIX"**, Winter USENIX: pages 29–41, January 1993.

[29] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse and H. van Stavern, **"Amoeba A Distributed Operating System for the 1990s"**, IEEE Computer, pages 44-51, January 1990.

[30] W. Rosenberry, D. Kenny, and G. Fisher, **"Understanding DCE"**, O'Reilly and Associates, Inc., 1992.

[31] Sun Microsystem, Inc**. "Programming Utilities & Libraries", chapter 2 "Lightweight Processes",** March 1990.

[32] ParaSoft Corporation, **"Express Building Parallel and Distributed Programs (Version 3.2)"**, 1992.

[33] A.S. Tanenbaum, **"Modern Operating Systems"**, pages: 507–524, 644–647, Prentice Hall, Englewood Cliffs, N.J., 1992.

[34] Thinking Machines Corporation, **"Connection Machine CM-5, Technical Summary"**, Cambridge, Massachusets, November 1992

Appendix A TCE Interface

Data Types

- TCE_Node — represents a computational node structure.
- TCE_Thrd — represents a thread structure.
- TCE_Sem — represents a semaphore structure.
- TCE_Port — represents a port structure.
- TCE_Chn — represents a channel structure.
- TCE_ChnSet — represents a channel set structure.
- TCE_ExcMsg — represents an exception message structure.

Predefined instances of the TCE data types

- TCE_InPort — standard receiving port.
- TCE_OutPort — standard sending port.
- TCE_ExcPort — standard exception port.
- TCE_NullPort — standard null port.

Predefined Constants

- Data Formats
  - TCE_FMT_Byte
  - TCE_FMT_Char
  - TCE_FMT_Uchar
  - TCE_FMT_Short
  - TCE_FMT_Int
  - TCE_FMT_Uint
  - TCE_FMT_Long
  - TCE_FMT_Float
  - TCE_FMT_Double
  - TCE_FMT_Node
  - TCE_FMT_Thrd
  - TCE_FMT_Sem
  - TCE_FMT_Port
  - TCE_FMT_Chn
  - TCE_FMT_ChnSet
  - TCE_FMT_Msg
- Architectures
  - TCE_ARCH_DontCare
  - TCE_ARCH_Alpha
  - TCE_ARCH_Sun4
  - TCE_ARCH_Mips

- ☐ Channel Characteristic
  - TCE_CHN_Reg — regular message passing mode.
  - TCE_CHN_Sync — synchronous message passing mode.
  - TCE_CHN_Force — force type of message passing.
  - TCE_CHN_AckThrd — acknowledgment generated when a message received by a thread.
  - TCE_CHN_AckNode — acknowledgment will be sent back when a message reaches a destination node.
  - TCE_CHN_Tag — a channel tag.
- ☐ Channel Set Characteristic
  - TCE_CHNSET_Tag — channel set tag.

Interface Calls

- ☐ tce_init — initializes the TCE environment.
- ☐ tce_exit — cleans the environment and exits with a specified return value
- ☐ tce_node — allocates the requested number of nodes.
- ☐ tce_exec — places processes on specified nodes and starts them.
- ☐ tce_rank — returns a rank within a given group of nodes, ports, channels and channel sets.
- ☐ tce_malloc — atomically allocates the requested number of bytes.
- ☐ tce_free — atomically frees a specified memory region.
- ☐ tce_time — returns the system, user and total times in millisecond units.
- ☐ tce_thrd_init — creates a thread, initializes its stack and its descriptor and returns a handle to it.
- ☐ tce_thrd_kill — unconditionally kills a thread.
- ☐ tce_thrd_start — puts a thread into its scheduling queue.
- ☐ tce_thrd_stop — removes a thread from its scheduling queue.
- ☐ tce_thrd_self — returns a thread handle to the requesting thread.
- ☐ tce_thrd_id — returns a thread id for a given thread handle.
- ☐ tce_thrd_priority — changes priority for the submitted thread.
- ☐ tce_thrd_yield — yields control to another thread.
- ☐ tce_thrd_join — blocks a calling thread when a specified child thread has not terminated yet.
- ☐ tce_thrd_detach — terminates the parent-child relation.
- ☐ tce_thrd_stack — sets a new default stack size.
- ☐ tce_thrd_timeout — blocks until the specified timeout interval expires
- ☐ tce_preempt_on — switches to the preemptive mode of execution.
- ☐ tce_preempt_off — switches back to the non-preemptive mode.
- ☐ tce_signal_on — enables signals (interrupts).
- ☐ tce_signal_off — disables signals.
- ☐ tce_clock_tick — sets a millisecond interval for the clock tick.
- ☐ tce_clock_slice — defines a time slice for a specified priority queue (in clock ticks).
- ☐ tce_sem_init — creates a semaphores and initializes it to a given value.
- ☐ tce_sem_close — destroys the semaphore structure.
- ☐ tce_sem_down — acquires a semaphore.
- ☐ tce_sem_up — releases a semaphore.
- ☐ tce_sem_updown — atomically acquires the first semaphore and releases the other.
- ☐ tce_port_get — returns a sending port to a submitted receiving port.
- ☐ tce_port_close — destroys a sending port.
- ☐ tce_port_register — registers a port under a particular name.
- ☐ tce_port_remove — removes a registered port.
- ☐ tce_port_request — returns a handle to a registered port.

- tce_chn_init — returns newly created channels.
- tce_chn_close — destroys specified channels.
- tce_chn_snd — sends a message through a channel.
- tce_chn_rcv — receives a message from a channel.
- tce_chn_set — imposes characteristics upon channels.
- tce_chnset_init — returns a new channel set, spanned over a vector of channels.
- tce_chnset_close — removes specified channels from a channel set (closes the set when all participating channels listed).
- tce_chnset_sync — performs a barrier operation on a particular set of channels.
- tce_chnset_bcast — makes a collective broadcast operation.
- tce_chnset_combine — collective combine, requires a user supplied function to perform a desired calculation.
- tce_chnset_concat — collective concatenation.
- tce_chnset_gather — collective gather.
- tce_chnset_prefix — collective prefix, uses a user supplied function for calculations.
- tce_chnset_reduce — performs a collective reduce using a user supplied function for doing calculations.
- tce_chnset_scatter — collective scatter.
- tce_chnset_shift — implements circular and regular shift operations.

Appendix B Matrix Multiplication — Multithreaded Implementation in TCE

```
#include <tce/tce.h>

#define SlaveName "/home/users/janusz/mm_rr.c"

#define NN 8
#define M 512

double A[M/NN][M];
double B[M/NN][M];
double C[M/NN][M];
double V0[2][M];
double *Vt;

int rank;

TCE_Node *Nodes[NN];
TCE_Port *RPorts[NN],*SPorts[NN],*Port;
TCE_Chn  *Chns[NN],*Chn;
TCE_ChnSet *ChnSet;
TCE_Sem  *End,*RcvStart,*RcvEnd;


void
send(TCE_Chn **chnp)
{
     int i;

     for(i = 0; i < M/NN; i++)
       tce_chn_snd(*chnp,&B[i][0],M,TCE_FMT_Double);
     tce_sm_up(End);
}

void
receive()
{
     int nnum,j,l,cpn = M/NN;
     int start = (rank + 1)*cpn;
     int end = start + M - cpn;
     double *Ptr;
     for(l = start; l < end; l++)
     {
       tce_sem_down(RcvStart); /* wait until previous column used */
       j = l%M;
       nnum = j/cpn;
       Ptr = &V0[l%2][0];     /* take a free vector */
       tce_chn_rcv(Chns[nnum],Ptr,M,TCE_FMT_Double);
       Vt = Ptr;
       tce_sem_up(RcvEnd);     /* notify that column received */
     }
}
void
compute_local()
{
     int i,j,k,t = 0;
     int start = rank*M/NN;
     int end = start + M/NN;
     for(j = start; j < end; j++,t++)
    for(k = 0; k < M/NN; k++)
    {
       for(i = 0; i < M; i++)
           C[k][j] = C[k][j] + A[k][i]*B[t][i];
       tce_thrd_yield();
    }
     tce_sem_up(End);
}
```

```
void
compute_remote()
{
     int i,j,k,l,cpn = M/NN;
     int start = (rank + 1)*cpn;
     int end = start + M - cpn;
     O_Thread *Thrd;

     RcvStart = tce_sem_init(1);
     RcvEnd = tce_sem_init(0);
     Thrd = tce_thrd_init(receive,(void *)0);
     tce_thrd_priority(Thrd,2);/* higher priority */
     tce_thrd_start(Thrd);     /* start the helping thread */
     tce_thrd_yield();
     for(l = start; l < end; l++)
     {
       j = l%M;
       tce_sem_down(RcvEnd);  /* request the current column */
       tce_sem_up(RcvStart);  /* let receive the next column */
       tce_thrd_yield();
       for(k = 0; k < M/NN;k++)
         for(i = 0; i < M; i++)
           C[k][j] = C[k][j] + A[k][i]*Vt[i];
     }
     tce_thrd_join(Thrd);
     tce_sem_up(End);
}
void
do_work()
{
     TCE_Thrd *Thrd;
     int i;

     tce_chn_set(Chns,TCE_CHN_Sync,1,NN);
     Thrd = tce_thrd_init(compute_local,(void *)0);
     tce_thrd_start(Thrd); /* gets the same priority 0 */
     Thrd = tce_thrd_init(compute_remote,(void *)0);
     tce_thrd_priority(Thrd,1); /* gets higher priority */
     tce_thrd_start(Thrd);
     for(i = 0; i < NN; i++)
       if(i != rank)
       {
         Thrd = tce_thrd_init(send,&Chns[i]);
         tce_thrd_priority(Thrd,3); /* highest priority */
         tce_thrd_start(Thrd);
       }
     /* tce_preempt_on() */
     End = tce_sem_init(-NN);
     tce_sem_down(End);    /* wait for all started threads */
}
void
master()
{
     TCE_Thrd *Thrd;
     TCE_Port **RemPorts = &SPorts[1];
     int i,n;

     /* allocate NN -1 nodes */
     if((n = tce_nodes(Nodes,(char **)0,TCE_ARCH_Alpha,NN - 1)) != NN - 1)
       tce_exit(1);
     /* place processes onto nodes */
     if((n = tce_exec(RemPorts,Nodes,SlaveName,TCE_InPort,n)) != NN - 1)
       tce_exit(1);
     rank = 0;
     for(i = 1; i < NN; i++)
       RPorts[i] = TCE_InPort;
     RPorts[0] = TCE_NullPort;
     SPorts[0] = TCE_NullPort;
     /* create NN channels */
     n = tce_chn_init(Chns,RPorts,SPorts,NN);
     SPorts[0] = TCE_InPort;
     /* send all collected sending ports to others */
     for(i = 1; i < NN - 1; i++)
        tce_chn_snd(Chns[i],SPorts,n,TCE_FMT_Port);
```

```
     /* create a channel set */
     n = tce_chnset_init(&ChnSet,Chns,NN);

     /* enforce the synchronous mode on all channels */
     tce_chn_set(Chns,TCE_CHN_Sync,1,NN);

     Thrd = tce_thrd_init(do_work,(void *)0);
     tce_thrd_start(Thrd);
     tce_thrd_join(Thrd);

     /* wait for other processes */
     tce_chnset_sync(ChnSet);

}
void
slave()
{
     TCE_Thrd *Thrd;
     int i,n;

     tce_chn_init(&Chn,&TCE_InPort,&Port,1);
     /* receive sending ports to other processes */
     n = tce_chn_rcv(Chn,SPorts,NN,FMT_Port);

     rank = tce_rank(SPorts,TCE_InPort,n,TCE_FMT_Port);
     for(i = 0; i < NN; i++)
       RPorts[i] = TCE_InPort;

     RPorts[rank] = TCE_NullPort;
     Chns[0] = Chn;
     tce_chn_init(&Chns[1],&RPorts[1],&SPorts[1],NN - 1);
     tce_chnset_init(&ChnSet,Chns,NN);
     tce_chn_set(Chns,TCE_CHN_Sync,1,NN);

     Thrd = tce_thrd_init(do_work,(void *)0);
     tce_thrd_start(Thrd);
     tce_thrd_join(Thrd);

     tce_chnset_sync(ChnSet);
}

void
main(int argc, char *argv[])
{
    TCE_Thrd *Thrd;

    if(!(Port = tce_init()))
      Thrd = tce_thrd_init(master,0);
    else
      Thrd = tce_thrd_init(slave,0);

    tce_thrd_start(Thrd);
    tce_thrd_join(Thrd);
}
```