A Study of Software Multithreading in Distributed Systems *

T.A. Marsland and Yaoqing Gao
Computing Science Department
University of Alberta
Edmonton, Canada T6G 2H1
<tony, gaoyq>@cs.ualberta.ca

Francis C.M. Lau
Computer Science Department
University of Hong Kong
Hong Kong
fmclau@cs.hku.hk

Technical Report TR 95-23
November 20, 1995

Abstract

Multiple threads can be used not only as a mechanism for tolerating unpredictable communication latency but also for facilitating dynamic scheduling and load balancing. Multithreaded systems are well suited to highly irregular and dynamic applications, such as tree search problems, and provide a natural way to achieve performance improvement through such new concepts as active messages and remote memory copy. Although already popular in single-processor and shared-memory processor systems, multithreading on distributed systems encounters more difficulties and needs to address new issues such as communication, scheduling and migration between threads located in separate addressing spaces. This paper addresses the key issues of multithreaded systems and investigates existing approaches for distributed concurrent computations.

Keywords: Multithreaded computation, thread, virtual processor, thread safety, active message, data-driven.

*This research was supported by Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Distributed computing on interconnected high performance workstations has been increasingly prevalent over the past few years, and provides a cost-effective alternative to the use of supercomputers and massively parallel machines. Much effort has been put into developing distributed programming systems such as PVM [20], P4 [8] MPI [31] and Express[1]. Most existing distributed software systems support only a process-based message-passing paradigm, i.e., processes with local memory that communicate with each other by sending and receiving messages. This paradigm fits well on separate processors connected by a communication network, where a process has only a single thread of execution, i.e., it is doing only one thing at a time. However, the process based paradigm suffers performance penalty through communication latency and high context switch overhead, especially in the case where computing work is frequently created and destroyed. To overcome these problems, lightweight processes or threads (subprocesses in a sheltered environment and having a lower context switch cost) have received much attention as a mechanism for (a) providing a finer granularity of concurrency; (b) facilitating dynamic load balancing; (c) making virtual processors independent of the physical processors; and (d) overlapping computation and communication to tolerate communication latency. In addition, multithreading provides a natural way to implement nonblocking communication operations, active messages and remote memory copy.

Multithreading can be supported by software or even hardware (e.g., MTA [3], Alewife [2], EARTH [23], START [24]). Although it has been popular in single-processor and shared-memory processor systems, multithreading on distributed systems encounters more difficulties mainly because of the high latency and low bandwidth communication links underlying distributed-memory computers.

In this paper, we first address some major issues about designing and implementing multithreading in distributed systems. Then we investigate typical existing multithreaded systems for exploiting the potential of multicomputers and workstation networks, and compare their capabilities. To provide a framework we use a uniform terminology of thread, process and processor, but provide at the end a table showing the varied terminology used by the originating researchers. From this study we identify three areas where further work is necessary to advance the performance of

multithreaded distributed systems.

2 Major issues

A runtime system defines the compiler's or user's view of a machine. It can be used either as a compiler target or for programmer use. It is responsible for management of computational resources, communication, and synchronization among parallel tasks in a program. Designing and implementing multiple threads in distributed systems involves the following major issues: (1) how to implement threads? (2) How to provide communication and scheduling mechanisms for threads in separate addressing space? (3) What kinds of programming paradigms are supported for ease of concurrent programming? (4) How to make the system thread-safe?

2.1 Threads

A process may be defined loosely as an address space together with a current state consisting of a program counter, register values, and a subroutine call stack. A process has only one program counter and does one thing at a time (single thread). As we know, one of the obvious obstacles to the performance of workstation networks is the relatively slow network interconnection hardware. One of the motivations to introduce the notion of threads is to provide greater concurrency within a single process. This is possible through more rapid switching of control of the CPU from one thread to another, because memory management is simplified. When one thread is waiting for data from another processor, other threads can continue executing. This improves performance if the overhead associated with the use of multiple threads is less than the communication latency masked by computation. From the view point of programmers, it is desirable for reasons of clarity and simplicity that programs are coded using a known number of virtual processors irrespective of the availability of the physical processors. Threads support a scenario where each process is a virtual processor and multiple threads are running in the context of a process. The feature of low overhead for thread context switch is especially suitable for highly dynamic irregular problems where threads are frequently created and destroyed.

Threads can be supported

- at the system-level, where all functionality is part of the operating system kernel;
- by user-level library code, where all functionality is part of the user program and can be linked in; and
- by a mixture of the above;

User-level library implementations such as Cthread [12], PCR [26], FastThreads [32],and pthread [30], multiplex a potentially large number of user-defined threads on top of a single kernel-implemented process. This approach can be more efficient than relying on operating system kernel support, since the overhead of entering and leaving the kernel at each call, and the context switch overhead of kernel threads is high. User-level threads are on the other hand not only flexible but can also be customized to the needs of the language or user without kernel modification. However user-level library implementation complicates signal handling and some thread operations such as synchronization and scheduling. Two different scheduling mechanisms are needed, one for kernel-level processes and the other for user-level threads. User-level threads built on top of traditional processes may exhibit poor performance and even incorrect behavior in some cases, when the operating system activities such as I/O and page faults distort the mapping from virtual processes to physical processors. The kernel thread support such as *Mach* [5], *Topaz* [9] and *V* [14] simplifies control over thread operations and signal handling, but, like traditional processes, these kernel threads carry too much overhead. A mixture of the above can combine the functionality of kernel threads with the performance and flexibility of user-level threads. The key issue is to provide a two-way communication mechanism between the kernel and user threads so that they can exchange information such as scheduling hints. The *Psyche* system [27] provides such a mechanism by the following approaches: shared data structures for asynchronoous communication between the kernel and the user; software interrupts to notify the user level of some kernel events; and a scheduler interface for interactions in the user space between dissimilar thread packages. Another example is *Scheduler Activations* [4], through which the kernel can notify the thread scheduler of every event affecting the user, and the user can also notify of the subset of user-level events affecting processor allocation decisions. In addition, *Scheduler Activations* address some problems not handled by the *Psyche* system such as page faults and upward-compatible simulation of traditional kernel threads.

2.2 Thread safety

A software system is said to be thread-safe if multiple threads in the system can execute successfully without data corruption and without interfering with each other. If a system is not thread safe, then each thread must have mutually exclusive access to the system, forcing serialization of other threads. To support multithreaded programming efficiently, a system needs to be carefully engineered in the following way to make it thread-safe: (1) use as few global state information as possible; (2) explicitly manage any global state that cannot be removed; (3) use reentrant functions that may be safely invoked by multiple threads concurrently without yielding erroneous results, race conditions or deadlocks; and (4) provide atomic access to non-reentrant functions.

2.3 Thread scheduling

Scheduling plays an important role in distributed multithreaded systems. There are two kinds of thread scheduling: scheduling within a process and among processes. The former are similar to the priority-based preemptive or round robin strategies in traditional single-processor systems. Here we will focus on more complicated thread scheduling among processes, especially for a dynamically growing multithreaded computation. Although some compilers can cope with the scheduling issues statically[28], their scope is often limited due to the complexity of the analysis and the lack of runtime information. For efficient multithreaded computations of dynamic irregular problems, runtime-supported dynamic scheduling algorithms can meet the following requirements: (1) provide enough active threads available to prevent a processor from sitting idle; (2) limit resource consumption, i.e., the total number of active threads is kept within the space constraint; (3) provide computation locality: try to keep related threads on the same processor to minimize the communication overhead.

There are typically two classes of dynamic scheduling strategies: work sharing (sender-initiated) and work stealing (receiver-initiated). In the work sharing strategy, whenever a processor generates new threads, the scheduler decides whether to migrate some of them to other processors on the basis of the current system state. Control over the maintenance of the information may be centralized in a single processor, or distributed among the processors. In the work stealing strategy, whenever

processors become idle or under utilized, they attempt to steal threads from more busy processors. Many work sharing and work stealing strategies have been proposed; For example, Chowdhury's greed load sharing algorithm [11], Eager et al.'s adaptive load sharing algorithm [15], and Karp et al.'s work stealing algorithm [25]. These algorithms vary in the amount of system information they use in making a task migration decision. Usually the work stealing strategy outperforms the work sharing strategy, since if all processors have plenty of work to do, there is no thread migration in the work stealing strategy, while threads are always migrated in the work sharing strategy. Eager et al. [16] have shown that the work sharing strategy outperforms the work-stealing at light to moderate system loads, but the latter is preferable at high system loads when the costs of task transfer under the two strategies are comparable.

2.4 Multithreaded programming paradigms

A programming paradigm provides a method for structuring programs to reduce the programming complexity. Multithreaded systems can be classified into shared-memory, explicit message-passing, active message-passing, and implicit data-driven programming paradigms. Multiple threads of control are popular in uniprocessors, shared-memory multiprocessors and tightly-coupled distributed memory multicomputers with the support of distributed shared memory systems, where a process has multiple threads and each thread has its own register set and stack. All the threads within a process share the same address space. Shared variables are used as a means of communication among threads. These systems support the shared-memory parallel programming paradigm: an application consists of a set of threads which cooperate with each other through shared global variables, as Figure 1(a) shows.

Most loosely-coupled multicomputers and networks of workstations are process based. Their software supports only message passing as the means of communication between processors, because there are no shared global variables. When threads are introduced to these systems, they are able to draw on the performance and concurrency benefits from multithreaded computing, but usually a shared address space is precluded (see Figure 1(b)).

Multithreads increase the convenience and efficiency of the implementation of active messages [33]. An active message contains not only the data but also the ad-

dress of a user-level handler (code segment) which is executed on message arrival. It provides a mechanism to reduce the communication latency and an extended message passing paradigm as well, as illustrated in Figure 1(c). There are fundamental differences between remote procedure calls and active messages. The latter has no acknowledgment or return value from the call. Active messages are also used to facilitate remote memory copying, which can be thought of as part of the active message paradigm, typified by *put* and *get* operations on such parallel machines as CM-5, nCUBE/2 [33], Cray T3D and Meiko CS-2 [22].

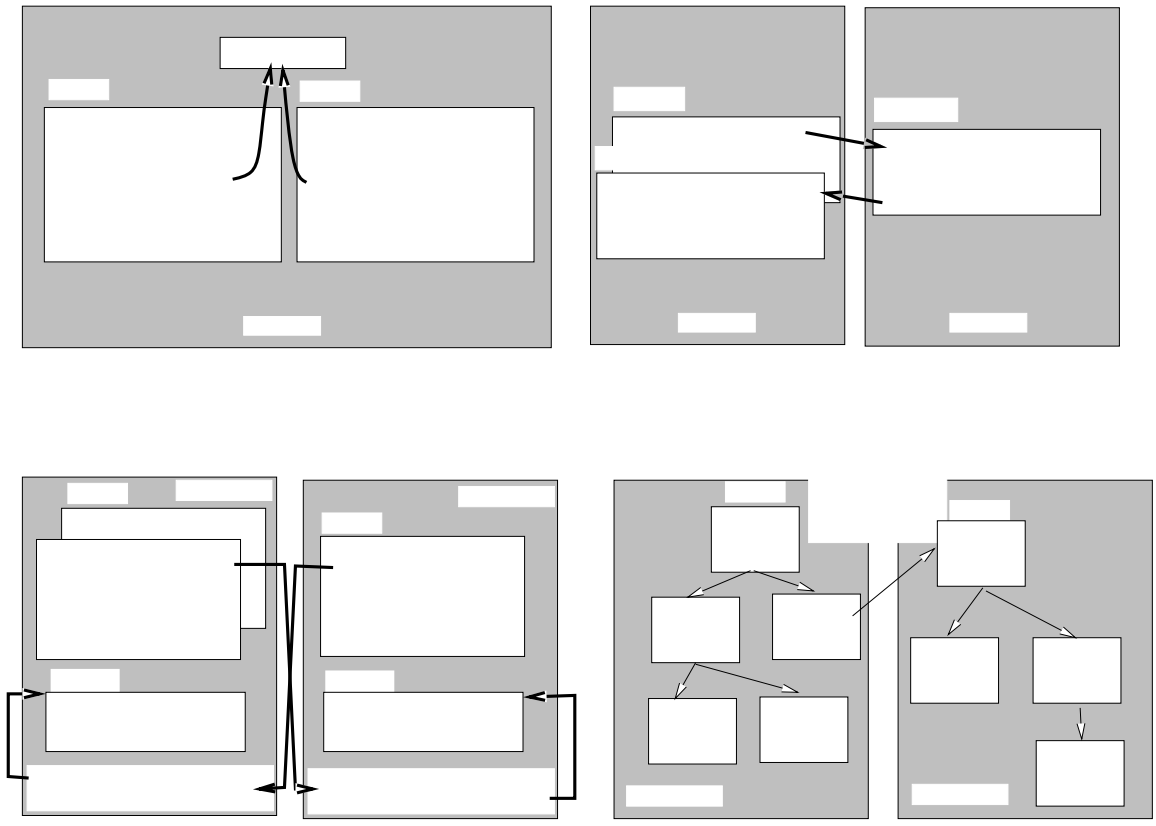

Figure 1: Multithreaded programming paradigms

In addition to the traditional shared-memory, message passing and active message parallel programming paradigms, there could be a data-driven non-traditional paradigm, see Figure 1(d), which is somewhat related to dataflow computing. There

an application consists of a collection of threads, but with data dependencies among the threads. A thread can be activated only when all its data are available. This paradigm simplifies the development of distributed concurrent programs by providing implicit synchronization and communication.

3 Multithreaded runtime systems

Since the varied and conflicting terminology is confusingly used in the literature, below we will compare different system using uniform terminology for continuity.

3.1 Cilk

Cilk [6] is a C-based runtime system for multithreaded parallel programming and provides mechanisms for thread communication, synchronization, scheduling as well as primitives callable within Cilk programs. It is developed by a group of researchers at MIT Laboratory for Computer Science.

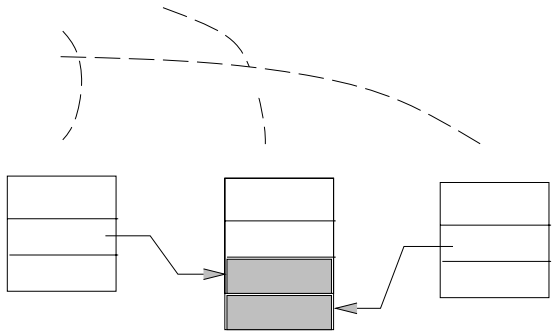

Figure 2: Threads, closures and continuations in Cilk

The Cilk language provides both procedure abstraction in implicit continuation-passing style and thread abstraction in explicit continuation-passing style. A continuation describes what to do when the currently executing computation terminates.

The programming paradigm provided by Cilk is a dataflow-like procedure-call tree, where each procedure consists of a collection of threads. The entire computation can be represented by a directed acyclic graph of threads. A thread is a piece of code, implemented as a nonblocking C function. When a thread is spawned, the runtime system allocates a closure (thread control table entry) for it. A closure is a data structure that consists of a pointer to the thread's code, all argument slots needed to enable the thread, and a join count indicating the number of missing arguments that the thread depends on. The Cilk preprocessor translates the thread into a C function of one argument (a pointer to a closure) and void return type. Each thread is activated only after all its arguments are available, i.e., its corresponding closure is changed from the waiting state to the full state. It runs to completion without waiting or suspending once it has been activated. A thread generates parallelism at runtime by spawning a child thread , so that the calling thread may execute concurrently with its child threads; For example, see Figure 2. The creation relationships between threads form a spawn tree. Threads may wait for some arguments to arrive in the future. Explicit continuation-passing is used as a communication mechanism among threads. A continuation is a global reference to an empty argument slot that a thread is waiting for.

Cilk's scheduler employs a work stealing strategy[7] for load balancing. Each processor maintains a local ready queue to hold full closures (threads whose data are available). Each thread has a level which is a refinement of the distance to the root of the spawn tree. A processor always selects a thread with the deepest level to run from its local ready queue. When a processor runs out of work, it selects a remote processor randomly and steals the shallowest ready thread from the remote processor based on the heuristic that a shallower thread is likely to have more heavy work than a deep one. Computations in Cilk are fully strict, i.e., a procedure only sends values to its parent. This reflects the ordinary procedure call-return semantics and has good time and space efficiency. Cilk also allows the programmer to have control over the runtime system.

Preliminary versions of the Cilk system has been implemented on massively parallel machines (CM5 MPP, the Intel Paragon MPP and the Silicon Graphics Power Challenge SMP) and networks of workstations (the MIT Phish). It has demonstrated good performance for dynamic, asynchronous, tree-like, MIMD computations, but not

yet for more traditional data-parallel applications. The explicit continuation-passing style results in a simple implementation but it needs more frequent copying of arguments and puts some burden on the programmer. Although the procedure abstraction with implicit continuation-passing is provided to simplify parallel programming, it is less good at controlling the runtime system.

3.2 Multithreaded Parallel Virtual Machine

PVM [20] has emerged as one of the most popular message-passing systems for workstations mainly because of its inter-operability across networks and the portability of its TCP- and XDR-based implementation. PVM only supports the process-based message-passing paradigm, where processes are both the unit of parallelism and of scheduling. TPVM and LPVM are two PVM-based runtime systems. They aim at enhancing PVM by introducing threads to reduce task initiation and scheduling costs, to overlap computation and communication, and to obtain finer load balance.

3.2.1 Threads-oriented PVM

The TPVM (Threads-oriented PVM) system [17] is developed by Ferrari and Sunderam in University of Virginia and Emory University. It is an extension of the PVM system that models threads-based distributed concurrent computation: an application's components form a collection of instances (threads) that cooperate through message passing. These instances/threads are the basic units of parallelism and scheduling in PVM.

TPVM supports not only user-level threads, and provides both the traditional process-based explicit message passing, but also a data-driven paradigm. In the traditional interacting sequential process paradigm, threads execute in the context of disjoint address spaces, and message passing is the only means of communication between threads. This paradigm typically forces the programmer to explicitly manage all thread creation and synchronization in a program. To implement this paradigm, PVM processes are first spawned and registered with the PVM system. Then the PVM processes export entry points describing threads, as Figure 3 shows. Threads can be activated in appropriate locations by a thread startup mechanism, i.e., the *tpvm_spawn* primitive.

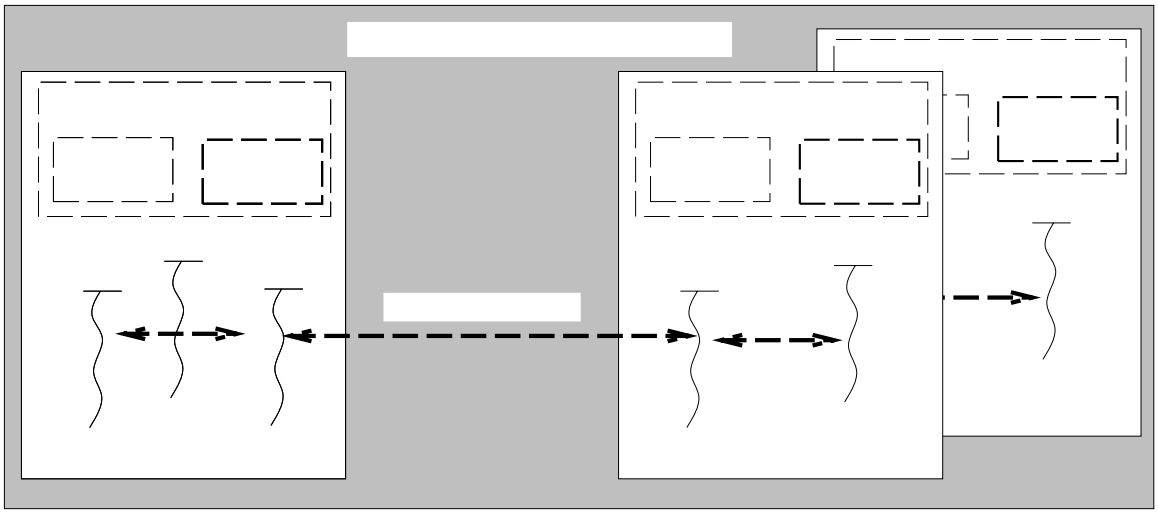

Figure 3: TPVM message-passing paradigm

The data-driven paradigm provides automatic synchronization and communication based on data dependencies. Threads should be declared with named data dependencies. They are initially created at appropriate locations, but are actually activated when a set of specified messages/data are available, as shown in Figure 4. As soon as a thread is activated, it can perform non-blocking receives to obtain its required inputs.

TPVM is implemented as three separate modules: (1) the library (the user interfaces with the TPVM system via library calls): the export functions (*tpvm_export, tpvm_unexport*), the message passing functions (*tpvm_send, tpvm_recv*), and the scheduling functions (*tpvm_spawn, tpvm_invoke*). (2) the portable thread interface module: thread creation, exiting the running thread, yielding the running thread, obtaining mutual exclusion, releasing mutual exclusion, and determining a unique identifier associated with the running thread. (3) the thread server module. A centralized thread server task/process provides support for the thread export database, scheduling, and data-driven thread creation. The scheduling of threads is done in a round robin fashion on the appropriate processes.

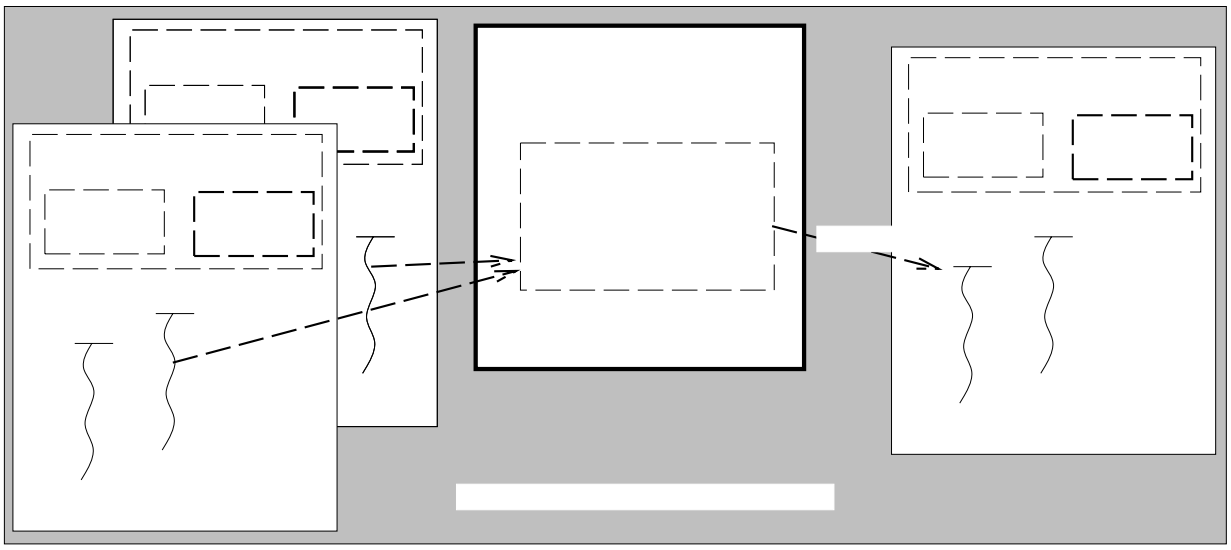

Figure 4: TPVM data-driven paradigm

Preliminary experiments [17] on a small network of Sparc+ computers show that some applications can benefit from TPVM's finer load balance, whereas many SPMD (Single-Program-Multiple-Data) style applications with very regular communication patterns do not.

TPVM is built on top of PVM and is portable. But the lack of thread-safety is still not addressed. As a result, the entire PVM library is treated as a critical region. Semaphores are used to ensure that only one active thread accesses the library at a time. Such an exclusive access requirement on the library is restrictive, e.g., the use of blocking PVM calls is precluded. The current version of TPVM manages threads through a centralized server and this may become a bottleneck as the number of processors increases.

3.2.2 Lightweight process PVM

LPVM (Lightweight process PVM) is developed by Zhou and Geist [34] at Oak Ridge National Laboratory. It is a modified version of the PVM system and is implemented on shared-memory multiprocessors.

In PVM, INET-domain UDP sockets are used to communicate among tasks. When one task sends a message to another, there are three copying operations involved: from

the sender's user space to the system's send buffer, to the system's receive buffer, and to the receiver's user space. With the availability of physical shared memory, the communication latency and bandwidth could be greatly improved. To retain the traditional PVM user interface, the whole user program in LPVM is one process with multiple tasks. Each task is a thread in LPVM instead of being a process in PVM. However it only supports the process-based message-passing paradigm.

To make LPVM thread safe, PVM was modified extensively to remove many global states [34]: (1) all the memory and buffer management code in version 3.3 of PVM was modified, and (2) "*pvm_spawn* " was changed so that the PVM library spawns new tasks, instead of the PVM daemon. The user interface is changed accordingly, e.g., most LPVM calls need to have the caller Task ID parameter and use it as a key to locate resources related to the task. "*pvm_pack*" routines explicitly require the send buffer id returned by "*pvm_initsend*" so that they can pack messages into the buffer safely.

LPVM is an experimental system and its current version runs on IBM SMP and SUN SMP systems with a single host. Threads are supported at the system level and thread scheduling within a process is fully taken care of by the underlying operating system. Thread scheduling among different hosts has not yet beed addressed in the current version, because it does not support the TPVM's "export" of the thread entry points to another host [17].

3.3 Nexus

The Nexus runtime system is developed at Argonne National Laboratory and California Institute of Technology [18]. It is designed primarily for a good compilation target to support task-parallel and mixed data- and task-parallel computation in heterogeneous environments.

Nexus supports the active message-passing paradigm. It provides five core abstractions: the processor, process, thread, global pointer, and remote service request. A processor is a physical processing resource. A process is an address space plus an executable program, which corresponds to a virtual processor. A computation consists of a set of threads. All threads in the same process communicate with each other through shared variables. One or more processes can be mapped onto a single

node. They are created and destroyed dynamically. Once created, a process cannot be migrated between nodes. A thread can be created in the same process as, or in a different process from, the currently executing thread. Nexus provides the operations of thread creation, termination, and yielding. To support a global name space for the compiler, a global pointer is introduced, which consists of three components: processor, process and address. Threads are created in a remote process (pointed to by a global pointer) by issuing a remote service request (RSR). Since Nexus does not require a uniform address space on all processors, RSR specifies the name rather that the address of the handler function. It is similar to an active message, but less restrictive. In a remote service request, data can be passed as an argument to a remote RSR handler by means of a buffer (see Figure 5).

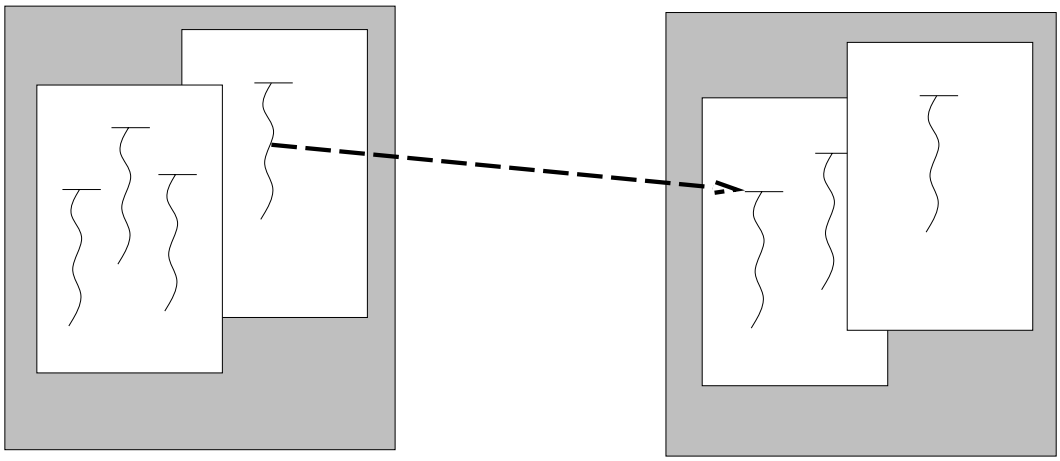

Figure 5: Threads, processes and processors in Nexus

The Nexus implementation encapsulates thread and communication functions in threads and protocol modules to support heterogeneity. Threads are implemented by a user-level library. It is the user's and compiler's responsibility to map computation to physical processors, which involves both the mapping of threads to processes and the mapping of processes to processors. Therefore Nexus is more suitable to a compiler target than to programmer use. Nexus is operational on TCP/IP networks of Unix workstations, the IBM SP1, and the Intel Paragon using NX. It has been used to implement two task-parallel programming languages: $CC++$ [10] and Fortran M [19]. Experiments [18] showed that Nexus is competitive with PVM despite its lack

of optimization.

3.4 Threaded Abstract Machine

TAM [13] is a Threaded Abstract Machine for fine-grain parallelism developed at University of California, Berkeley. It serves as a framework for implementing a general purpose parallel programming language. TAM provides efficient support for the dynamic creation of multiple threads of control, locality of reference through the utilization of the storage hierarchy and scheduling mechanisms, and efficient synchronization through data-flow variables. TAM supports a novel data-driven paradigm.

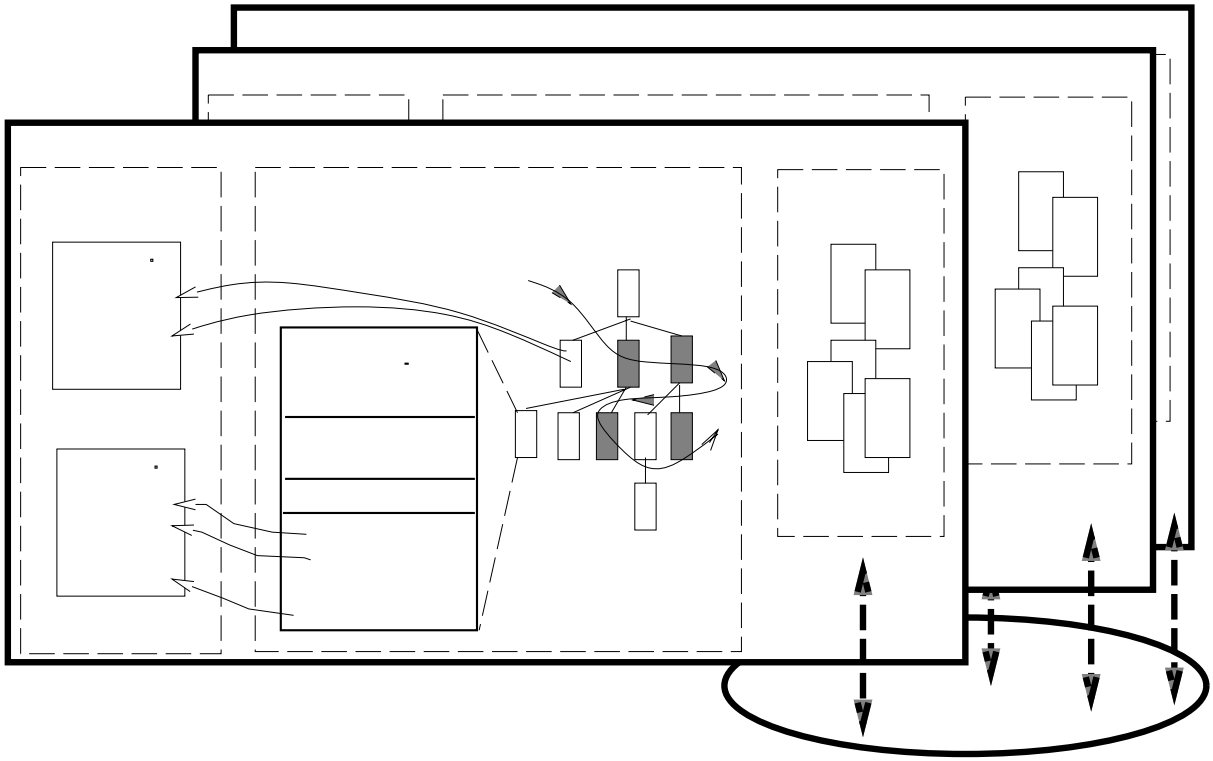

Figure 6: TAM structure

A TAM program is a collection of code-blocks and each code-block consists of several non-blocking and non-branching *instruction sequences* and inlets (message handlers). When a caller invokes its child code-block, it does not suspend itself while an activation frame (roughly comparable to a thread) is allocated for the child code-block. The activation frame provides storage for the local variables and the resources required for synchronization and scheduling of instruction sequences. The dynamic call structure forms a thread tree. A TAM instruction sequence runs in the context of an activation frame without any suspension and branching. Argument and result passing is represented in terms of inter-thread communication: the caller sends arguments to predefined inlets of the callee, who in turn sends results back to inlets the caller specifies. An inlet is a compiler-generated message handler which processes the receipt of a message for a target thread (see Figure 6).

Threads are the basic unit of parallelism between processors. A thread is ready to run if it has some enabled instruction sequences waiting to be executed. A thread has a continuation vector for the addresses of all its enabled instruction sequences. Each processor maintains a scheduling queue consisting of ready threads. Finer parallelism within a thread is represented by instruction sequences that are used to reduce communication latency. Furthermore parallelism within an instruction sequence can be used to improve processor pipeline latency.

TAM provides mechanisms to control the location of threads and the compiler may determine this mapping statically. For highly irregular parallel programs, the runtime system needs to provide dynamic load balancing techniques.

To validate the effectiveness of the TAM execution model, a threaded machine language called "TL0 for TAM" is implemented on the CM-5 multiprocessor. The fine grain parallelism achieved makes TAM especially suitable for tight-coupled multicomputers.

3.5 Multithreaded Message Passing Interface

MPI (Message Passing Interface) [31] tries to collect the best features of many existing message-passing systems and improve and standardize them. MPI is a library which encompasses point-to-point and collective message passing, communication scoping, virtual topologies, datatypes, profiling, and environment inquiry. It sup-

ports a process-based message-passing paradigm. Processes in MPI belong to groups. Contexts are used to restrict the scope of message passing. A group of processes plus the safe communication context form a communicator, which guarantees that one communication is isolated from another. There are two types of communicators: intra-communicators for operations within a single group of processes, and inter-communicators for point-to-point communication between two non-overlapping groups of processes.

Haines et al. at NASA Langley Research Center [21] developed a runtime system called Chant, which is layered on top of MPI and POSIX *pthreads*. In Chant, threads are supported at the user-level and pthreads are extend to support two global threads: *chanter* and *rope* (a collection of chanters). A global thread identifier is composed of a process identifier, a process rank within the group and a thread rank within the process. Communication operations resemble the MPI operations. Communication between threads within the same process can use shared memory primitives while communication between threads in different processes utilizes point-to-point primitives. In point-to-point communication, Chant employs user-level polling for outstanding messages. Like Nexus, Chant supports remote service requests for a remote fetch, remote state update and remote procedure calls. The existing pthreads primitives are extended to provide a coherent interface for global threads. Chant supports the message-passing and active message-passing paradigms. But it has not addressed the issue of thread scheduling in a different addressing space.

Chant is currently running on both a network of workstations and on the Intel Paragon multicomputer. It is being used to support Opus [29], a superset of High Performance Fortran (HPF), with extensions that support parallel tasks and shared data abstractions.

4 Conclusion

Threads are an emerging model for exploiting multiple levels of concurrency and computational efficiency in both parallel machines and networks of workstations. This paper investigates typical existing multithreaded distributed systems, using a uniform terminology for clarity. Table 1 summarizes the varied terminology of different research teams, and provides the association between the terms thread, process and

processor which we use. Compared to the single-processor and shared memory multiprocessor cases, multithreading on distributed systems encounters more difficulties due to underlying physical distributed memories, and the need to address new issues such as thread scheduling, migration and communication in a separate addressing space.

Table 1: Threads, processes and processors

Terms	Cilk	TPVM	LPVM	Nexus	TAM	Chant
Instruction sequence	instruction sequence	instruction sequence	instruction sequence	instruction sequence	thread	instruction sequence
Thread	thread, closure	thread	thread	thread	activation frame	chanter, rope
Process	–	task	task	context	–	process
Processor	processor	processor	host	node	processor	processor

The systems we discuss here are either built on top of such existing systems as PVM and MPI, through the integration of a user-level threads package, or are self-contained. Because of the nature of the underlying systems, usually there is no shared address space among threads, even if these threads belong to the same process. The multithreaded programming environments supported by these systems include explicit message passing, active message passing and implicit data-driven paradigms. Thread scheduling can be taken care of by the software systems themselves, or by the underlying operating system, or simply left to users and compilers.

TPVM supports both message passing and data-driven paradigms. Threads have no shared data even if they belong to the same PVM process. This makes it easier to activate a thread at any appropriate location. But it is obvious that communication between threads within the same process by message passing instead of shared memory is not natural and may result in extra overhead. In TPVM, the lack of thread and signal safety has not yet been addressed, and its centralized server may become a bottleneck for large applications. Modifications in LPVM make it thread and signal safe, and thread scheduling is completely taken care of by the operating system. To retain the user interface, LPVM supports only the message passing paradigm, i.e., threads cannot share data even though the underlying machines have physical shared

memories. The merit of LPVM's approach to combining PVM with multithreading on shared-memory multiprocessors remains unclear. LPVM reduces the communication latency by taking advantage of shared memory in a multiprocessor, but seems not to have yet addressed multithreading on multiple hosts.

Although most user thread packages, such as pthread, support only communication between threads located in separate address space, Chant extends the POSIX pthread by adding a new global thread (chanter) and provides point-to-point and remote service request communication operations by utilizing the underlying MPI system. Threads within the same process can share data. Furthermore Chant supports collective operations among thread groups using a scoping mechanism called ropes. Chant supports the message passing paradigm. In contrast, Nexus is a multithreaded system mainly used for a compiler target. It supports dynamic processor acquisition, dynamic address creation, a global memory model and remote request service. Nexus supports the active message passing paradigm. The burden of mapping threads to processes and processes to physical processors is left to users or compilers. Finally, Cilk is a C-based multithreaded system suitable for dynamical tree-like computations. It supports the data-driven paradigm and thread scheduling is automatically taken care of by the runtime system. Cilk has not yet shown its efficiency in data-parallel applications and its continuation-passing style results in extra copying overhead, but benefits from its work-stealing strategy. TAM also supports the data-driven paradigm. Compared with Cilk, it exploits even finer grain parallelism. Therefore it is more suitable for tightly coupled distributed memory multicomputers. Table 2 summarizes the important features of the systems discussed here.

For further advances to multithreading on distributed systems the following three topics should be addressed:

- Integration of multiple programming paradigms to support both data-parallel and task-parallel applications. Multithreading has demonstrated its suitability for task-parallel computations, but most existing multithreaded systems have not yet obtained good efficiency for data-parallel problems.

- Coordination between thread and process scheduling. In multithreading systems, both thread and process scheduling are supported. Little attention has been paid so far to coordinate these two activities for system-wide global load

balancing. Thread scheduling can be viewed a mechanism for short-term load balancing within a single user, while process scheduling helps long-term load balancing. One must therefore recognize when a workstation is disconnected or is newly available, and migrate processes.

- Fast communication mechanisms. The protocols of existing communication networks are heavyweight and the software overhead of communication dominates the hardware routing cost. Improvements in communication performance can dramatically increase the applicability of multithreading.

In our view, advances in each of these areas will have dramatic effect on the utility and performance of thread-based distributed computing systems.

Table 2: Capability Comparison of Multithreading Systems

Capabilities	Cilk	PVM based		Nexus	TAM	MPI based Chant
		TPVM	LPVM			
Parallel programming paradigms	data-driven	message-passing, data-driven	message-passing	active message-passing	data-driven	active and message-passing
Languages supported	Cilk: C-based language	C language	C language	Mainly used as a compiler target for pC++, HPF, FortranD	TL0: an abstract machine Id90	Opus: Fortran -based language
Thread implementation	user-level	user-level	system-level	user-level	user-level	user-level
Underlying software systems required	no special software system required	PVM without any modification, GNU/REX thread library	PVM with minimal modification for thread safety, OS-supported threads	POSIX, DCE, C, and Solaris threads, Various protocols: TCP,Intel NX	no special software system required	MPI, pthreads with extensions
Thread scheduling strategies	work stealing: select one processor at random	dynamic scheduling at thread creation or at the request of users	Scheduling taken care of by OS	taken care of by users or compilers	compiler controlled and runtime supported frame and thread scheduling	taken care of by users or compilers
Underlying architectures required	multi-processors, multi-computers, a cluster of workstations	hetero-geneous distributed environment	Shared memory multi-processors	heterogeneous distributed environments	tightly coupled multi-computers	multi-computers, a network of workstations
Strength	efficient tree-like computations, dynamic scheduling strategy,	supports two paradigms, portable on any PVM system	take advantage of shared memory, modify PVM for thread safety	global name space, support mixed parallel computations, remote service request	extended data-driven paradigm, compiler-controlled scheduling	asynchronous remote service requests,
Weakness	frequent argument copying overhead, not suitable for data parallel problems	use only subset of library due to thread safety, a bottleneck caused by the server	no support of thread migration from one host to another	no support of automatic scheduling strategy, not suitable for programming	only suitable for tightly coupled machines due to fine parallelism	users take care of thread scheduling

References

[1] The Express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken and six others. The MIT alewife machine: Architecture and performance. In *Proceedings of ISCA'95*, 1995.

[3] Robert Alverson, David Callahan, Daniel Cummings, and three others. The tera computer system. In *Proceedings of International Conference on Supercomputing*, pages 1–6, June 1990.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[5] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing,. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, NM, USA, November 1994.

[8] Ralph Butler and Ewing Lusk. User's guide to the P4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992. Version 1.4.

[9] Thacker C., Stewart L., and Satterthwaite Jr. E. Firefly: A multiprocessor workstation. *IEEE Trans. on Computers*, 37(8):909–920, August 1988.

[10] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Aki Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, Cambridge, MA 02142, 1994.

[11] S. Chowdhury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9(1):93–99, May 1990.

[12] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, 1994.

[13] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. In *Journal of Parallel and Distributed Computing, Special Issue on Dataflow*, June 1993.

[14] Cheriton D. The V distributed system. *Communications of ACM*, 31(3):314–333, March 1988.

[15] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[16] D.L. Eager, E.D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[17] Adam Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *IEEE High Performance Distributed Computing 4*, August 1995. pp. 211-218

[18] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task-parallel programming languages. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 1994.

[19] Ian T. Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, June 1992.

[20] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994.

[21] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of chant: A talking threads package. In *Proceedings of Supercomputing '94*, pages 350–359, 1994. Also as Tech report NASA CR-194903 ICASE Report No. 94-25, Institute for Computer Applications in Science and Engineering, NASA Langley Research.

[22] M. Homewood and M. McLaren. Meiko CS-2 interconnect elan - elite design. In *Proceedings of Hot Interconnects'93*, pages 2.1.1–4, August 1993.

[23] Herbert H.J. Hum, Olivier Maquelin, Kevin B. Theobald and fourteen others. A design study of the EARTH multiprocessor. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 59–68, June 1995.

[24] Micheal J. Beckerle. Overview of the START (*T) Multithreaded Computer. In *Proceedings of the IEEE COMPCON*, February 1993.

[25] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[26] Weiser M., Demers A., and Hauser C. The portable common runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, Litchfield Park, Ariz., 1989.

[27] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pages 110–121, October 1991.

[28] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[29] Piyush Mehrotra and Matthew Haines. An overview of the opus language and runtime system. Technical Report NASA CR-194921 ICASE Report No. 94-39, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, May 1994. 16 pages.

[30] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.

[31] Anthony Skjellum, Nathan E. Doss, Kishore Viswanathan, Aswini Chowdappa, and Purushotham V. Bangalore. Extending the message passing interface (MPI). In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference II*. IEEE Computer Society Press, October 1994.

[32] Anderson T., Lazowska E., and Levy H. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Trans. on Computers*, 38(12):1631–1644, December 1989.

[33] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[34] Honbo Zhou and Al Geist. LPVM: lightweight process (thread) based PVM for SMP systems. Technical report, Oak Ridge National Laboratory, 1995.