Runtime Support for User-Level Ultra Lightweight Threads on Distributed Memory Computers

Wei Shu[1]
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

Abstract. Ultra-lightweight Thread (uThread) is a library package designed and optimized for user-level management of parallelism in a single application program running on distributed memory computers. Existing process management systems incur an unnecessarily high cost when used for the type of parallelism exploited within an application. By reducing the overhead of ownership protection and frequent context switches, uThread encourages both simplicity and performance. In addition, uThread provides various scheduling support to balance the system load. The uThread package reduces the cost of parallelism management to nearly the lower bound. This package has been successfully running on most distributed memory computers, such as Intel iPSC/860, Touchstone Delta, NCUBE, and TMC CM-5.

1. Introduction

With the evolution of massively parallel systems many scalable homogeneous multiprocessor machines are distributed memory parallel computers which are designed for large-scale scientific, engineering, and business applications. Many application problems with uniform computation patterns have been successfully parallelized on these machines. On the other hand, it is difficult to parallelize application problems with nonuniform computation structures. Solving this class of problems requires efficient management of parallelism, as well as effective scheduling of parallel subcomputations. In this paper, we describe a runtime support system in solving a single application with irregular and dynamic features running on a distributed memory computer.

As a single application problem is divided into many partitions to be executed in parallel, the overhead associated with each partition should be minimal. In this special context, traditional

[1]This research was partially supported by NSF grants CCR-9109114.

processes are not appropriate vehicles to express parallelism, because many features of a process provided for general-purpose computation are often unused, incurring unwarranted costs. What we need is a simple notation to represent parallel actions and the way to support them. One such a notation is the thread, which is a lightweight process [24]. With the thread provided, solving an application in parallel involves execution of many dynamically created threads. Each thread runs strictly sequentially, and has its own program counter and stack to keep track of where it is. Different thread architectures have been investigated and developed [17, 3, 13].

The overhead associated with a thread is much lighter than that with a traditional process. However, the overhead associated with the context switching is still large. In a single application domain, it is possible to avoid context switching by using the *run-to-completion* (RC) thread [2]. Once a RC thread starts execution, it will run to completion without being blocked. Therefore, a RC thread may not need its own stack and may not need to save registers, and the overhead can be minimized. We propose a two-level structure based on the RC thread concept — the *ultra-lightweight Process (uProcess)* and the *ultra-lightweight Thread (uThread)*. A uProcess, which consists of one or more uThreads, is a unit for allocation. A uThread is an indivisible unit of execution, in other words, a RC thread. Execution of each uThread is driven by a message and will continue until its completion. The activities that may result in a blocking state include I/O operations, file operations, communication and synchronization primitives, etc. With a message-driven model applied, threads communicate and synchronize with each other via messages. Here, no receive operation is permitted, and all send operations must be non-blocking. Furthermore, applications applied in this context are assumed to be computation-intensive and do not have frequent file operations. All I/O and file operations can be performed within special threads that are dedicated to deal with I/O and file operations. The feature of run-to-completion also implies nonpreemptive scheduling. Since fairness is less of a concern within a single application, a nonpreemptive scheduling strategy can be realistic and effective.

A library package, named the uThread package, is designed and optimized for user-level management of parallelism in a single application program running on distributed memory computers. It supplies primitives to manipulate the parallelism within a program and provides a platform to support various scheduling strategies. The uThread package can be implemented at the ker-

nel level or user level. We do not implement the uThread package at the kernel level based on the following considerations: (1) Kernel-level implementations are not as efficient as user-level implementations. The kernel is usually required to provide more complete features, and therefore, incurs substantial overhead. The switching into and out of supervisor mode can also be of high cost. Especially when uProcesses are employed to exploit parallelism, the grain-size of a uProcess could be small and system primitives could be so frequently called that the overhead with user/supervisor mode switching can become unbearable. (2) Since applications running on parallel machines fall into a wide spectrum of computational models that need different system functionalities, it is almost impossible to design a kernel to satisfy all requirements. Instead, moving these functions to user-level can customize individual application characteristics and offer users flexibility. (3) It is technically feasible to implement the uThread package in the user space, since no change in existing operating system kernel is needed. Therefore, we believe that managing parallelism at the user level instead of the kernel level is essential in terms of efficiency and high-performance.

2. The uThread Package Overview

The uThread package is designed and implemented for solving irregular and dynamic problems. The computation can be dynamically divided into many subcomputations performed in parallel. Each subcomputation is realized by a uProcess. There is no commonly shared data available among uProcesses. All uProcesses communicate with each other via messages. Within a uProcess, there could be one or more threads of computation performed, each of which corresponds to a uThread. Different from uProcesses, all uThreads within the same uProcess can have a *shared data area (SDA)* and exchange information through the SDA. Figure 1 shows the general organization of uProcesses, uThreads, and SDAs.

The structure of uProcesses and uThreads is similar to that of processes and threads. But the execution of uProcesses and uThreads is message-driven. Every computation in this system, thereby every uThread, is the result of processing a message. New messages can be generated by a uThread. These unprocessed messages are the driving force behind computation in the system. Here, messages are *active* and uThreads are *passive* in the sense that a message will trigger the corresponding uThread to do computation. Furthermore, a uThread is an indivisible unit of

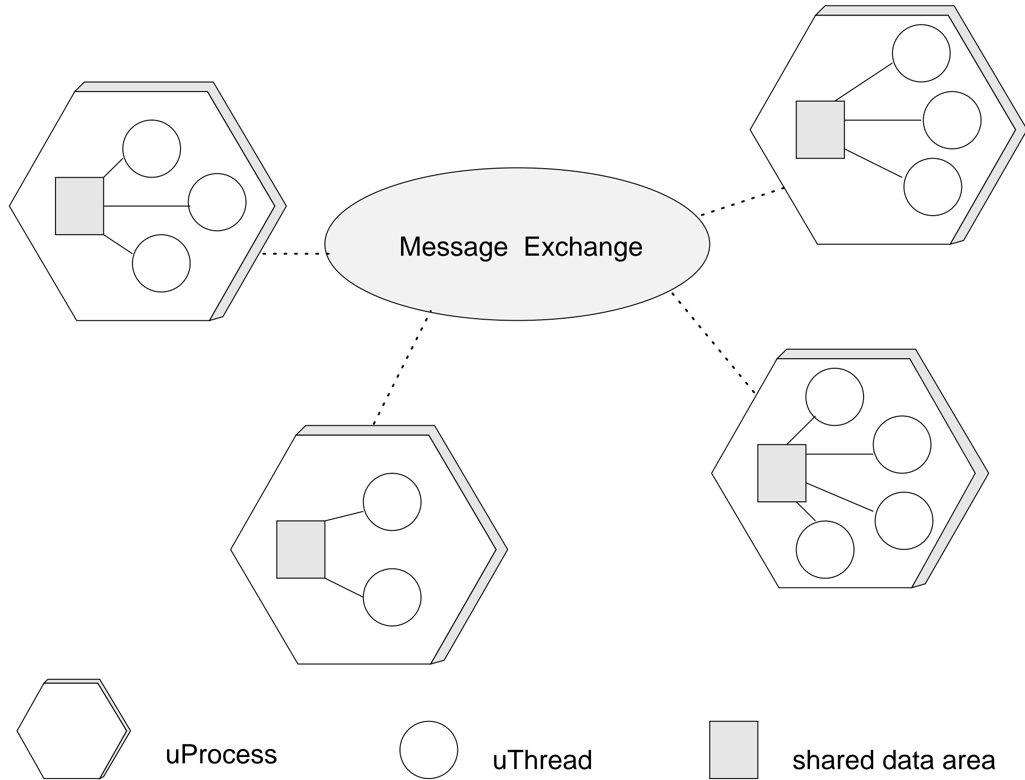

Figure 1: The Structure of uProcess, uThread, and SDA.

execution. Once it is triggered by a message, it runs continuously until completion.

In addition to one-to-one mapping of messages and uThreads, the creation of a uProcess can also be represented by a message, which triggers a *initial* uThread to start the uProcess environment, including allocation and initialization of its SDA, and perform some computation if needed. The model is not strictly message-driven since the uThreads within a uProcess can have the shared data persisting beyond the duration of an individual uThread execution.

Each message contains the address of a uThread instruction sequence and the data on which the uThread is to perform computation. Note that we assume the Single Program Multiple Data (SPMD) programming model, therefore, we rely on a uniform code image accessible at each processor. Messages are further classified into two types. One type of messages that is associated

with the creation of new uProcesses is called *uProcess messages*. The other type of messages that corresponds to the uThreads is called *uThread messages*. While a uThread message needs an extra attribute to indicate which uProcess it is destined for, a uProcess message requires an attribute to specify the SDA size of the uProcess to be created. Parallelism is exploited at the level of uProcesses, but not of uThreads. The existence of uThreads provides users with flexible control of multiple execution threads. In a uProcess, there is no presumed sequence to indicate which uThread will be carried out first. Execution of uThreads is totally dependent upon the arrival of messages.

There are two basic primitives to be supported: the creation of a new uProcess and the generation of a uThread message. The entire computation originates from a kernel-level process. It can create uProcesses at any time. It is able to perform any regular kernel-level operations, too, including I/O and file operations, and may be suspended or blocked. However, the code written for a uThread is restricted. It can perform computation, make any kernel-level non-blocking calls, and use the uThread system primitives provided. In addition, the code of a uThread can access the SDA of the uProcess without locking/unlocking, since there is no competition between any run-to-completion threads and no parallelism exploited between uThreads. In Figure 2, we list the major system primitives provided by the uThread package.

3. Design and Implementation

There have been several user-level thread packages developed; most of them were implemented on shared memory multiprocessors. With a shared memory available, all user-level threads within a single process share the same address space. As a result, they can have global variables and, thereafter, need mechanisms to protect the concurrent access to those shared variables. Furthermore, a thread can be executed by any processor, since all processors are within a single address space. Our design and implementation are based on the distributed memory multicomputers. The context is different in terms of memory organization. User-level uProcesses cannot share the same memory address space if they are distributed to different processors. A kernel-level process must be created at each processor to build up a substrate for uProcesses and uThreads. Each kernel-level process runs the user-level package and user application program, both of which are written in the SPMD style. Since there is no shared memory between these kernel-level processes,

- PID = uProcess_Create(uProcName, uThreadCode, data, dataSize, SDAsize, PIDflag)
  uProcName: name of the uProcess
  uThreadCode: pointer to the code of the initial thread
  data: pointer to the data to be used for the initial thread
  dataSize: length of the data
  SDAsize: length of the SDA for the new uProcess
  PIDflag: true if need a PID returned, otherwise false
  PID: returned uProcess identifier if PIDflag is true

- Send_Message(destPID, uThreadCode, data, dataSize)
  destPID: destination uProcess identifier
  uThreadCode: pointer to the code of the specified uThread
  data: pointer to the data to be used for the specified uThread
  dataSize: length of the data

- uProcess_Exit(PID)
  PID: uProcess identifier to be terminated

- PID = uProcess_Self()
  PID: returned uProcess identifier for itself

- PID = uProcess_Parent()
  PID: returned uProcess identifier for its parent uProcess

Figure 2: Major System Primitives for uThread Package.

coordination among them needs more sophisticated schemes. For simplicity, we assume that the entire distributed memory parallel machine is dedicated to a single application, or the machine is used in a timesharing mode with a group scheduling applied. Thus, at any time, we have N kernel-level processes running in parallel, asynchronously, where N is the number of processors in the machine.

Since we only exploit parallelism at the level of uProcesses, once a uProcess is scheduled to a particular processor, all its uThreads are forced to be executed at the same processor. No process migration is supported. Notice that the computation is message driven. Once a message arrives, the corresponding uThread is ready to execute. Therefore, we do not have a ready uProcess queue for the processor-level scheduling. For each processor, instead, there is a single *message queue* to

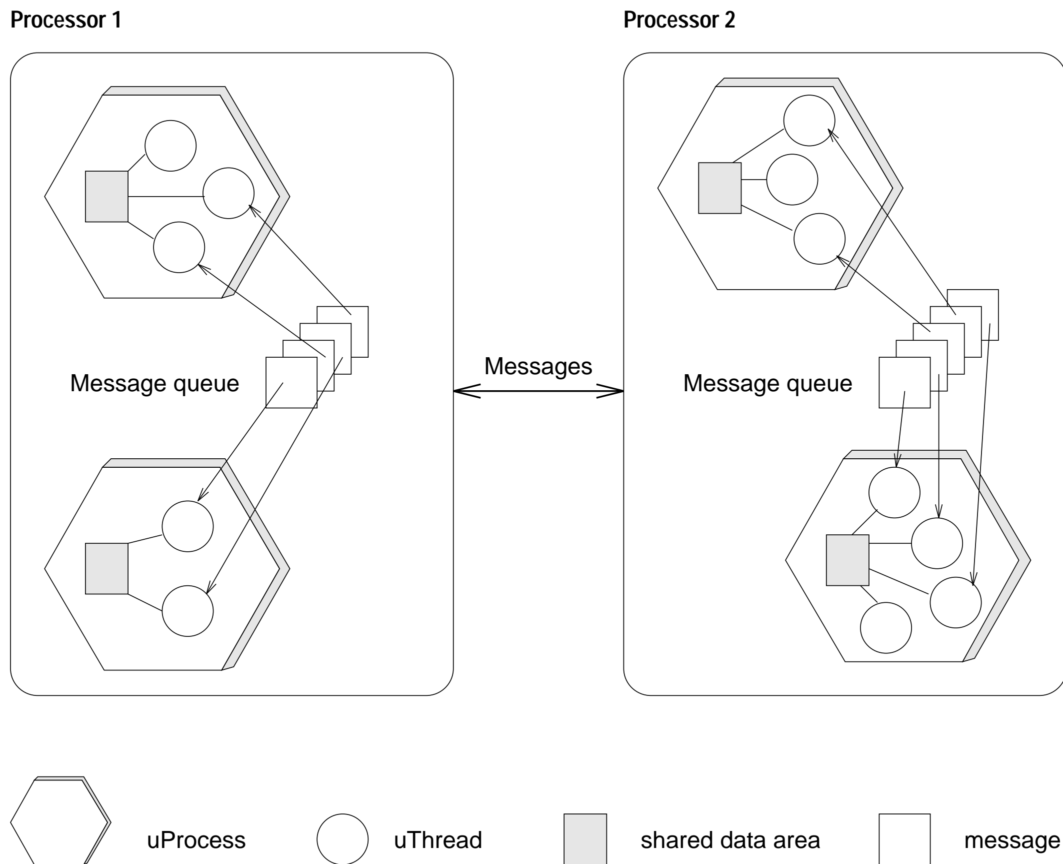

Figure 3: The uThread structure on distributed memory systems.

buffer all incoming messages. When a message reaches the head of the queue, its corresponding uThread is executed. Thus, in a distributed memory system, the organization of the uProcess, uThread, and SDA is illustrated in Figure 3.

3.1. The uProcess Creation and Identifier

The context of a uProcess is encapsulated in a process control block (PCB), as usual. Since uThreads are RC threads, there is no control block needed for a uThread. Information in a PCB includes the address to its SDA, the process status, its parent information, etc.

The creation of a uProcess in a distributed memory environment is different from a traditional case. It involves two major steps: (1) When the uProcess_Create() call is made, the uProcess cannot be created immediately because it is unknown where this new uProcess will be scheduled.

Instead, its creation request is generated as a uProcess message to be scheduled. At this time, the uProcess is said to be *virtually created*. (2) When a uProcess message is scheduled to a processor and reaches the head of the message queue for processing, the uProcess is *physically created* in the sense that the corresponding PCB is allocated and the initial uThread is executed. All active uProcesses in a processor are chained by their PCBs. A uProcess can terminate itself by calling uProcess_Exit(), which will take its physical PCB away from the resident uProcess chain and mark its state as *terminated*.

Every uProcess in the system is assigned a unique identifier (PID). Usually, a PID is implemented as a nonnegative integer, and a hash table or a one-to-one mapping table can be used to find its corresponding PCB. However, support of a global hash table in a distributed memory machine requires expensive communications. It is natural to integrate the processor number into a PID so that we can at least know to which processor the uProcess is allocated. Unfortunately, when a call uProcess_Create() is made, it is unknown where this new uProcess will be scheduled. Therefore, the PID assigned at that time cannot include the processor number yet. To solve this problem, we associate two types of identifiers with each uProcess: one called *virtual PID* and the other called *physical PID*.

virtual PID — When a uProcess is virtually created, if requested, the call of uProcess_Create() returns a virtual PID, which consists of the processor number where the creating uProcess resides and the address of a virtual PCB, which is created to establish a link to the future physical PCB later on. Notice that the virtual PID is included in the uProcess message that is to be scheduled to some processor for the physical creation.

physical PID — When a uProcess is physically created, its physical PID consists of the processor number where the new uProcess resides and the memory address where its physical PCB is allocated. At this time, if a virtual PID is present in the uProcess message, a *physical-virtual-link* message that includes the new physical PID is sent back to the processor indicated by the virtual PID. The physical PID in the message is then stored in the virtual PCB to establish a link between the virtual and physical PCBs.

With the virtual PID and PCB possibly introduced, the more detailed steps involved in

- VIRTUAL CREATION: when the uProcess_Create() call is made
  create a uProcess message
  if a virtual PID is requested
    allocate a virtual PCB, set its physical PID as *null*
    combine the current processor number and the address of this PCB
      to form a virtual PID to be returned
    include the virtual PID in the uProcess message
  pass the uProcess message to the scheduling module

- PHYSICAL CREATION: when a uProcess message is processed
  allocate a physical PCB and the shared data area
  combine the current processor number and the address of this PCB
    to form a physical PID to be stored in PCB
  if there is a virtual PID contained in this message
    generate a system control message, named as *physical-virtual-link* message,
      to send the physical PID back to its virtual PID
  link the newly created PCB to the resident uProcess chain
  execute the uThread code with the data and SDA passed to it

- LINK ESTABLISHMENT: when a physical-virtual-link message is processed
  find out the virtual PCB
  write the physical PID into the virtual PCB
  if there is any message in the pending queue of the virtual PCB
    send the pending messages to the new physical PID

Figure 4: Algorithms for uProcess Creation.

creation of a uProcess are shown in Figure 4. A virtual PCB is an intermediate spot between a virtual PID and the corresponding physical PCB. Before the arrival of physical-virtual-link message, the virtual PCB buffers all the pending messages. Once the physical PID becomes available, the virtual PCB serves as a redirection station. In this implementation, once a PCB is allocated, it cannot be released since the PID is directly associated with address of the PCB. This approach eliminates the mapping table from a PID to its PCB, but occupies the memory space of a PCB even after the corresponding uProcess is terminated. It is based on the fact that the PCB is small and the lifetime of all uProcesses is relatively short due to the single application context.

3.2. Sending Messages by Using PIDs

In our system, PIDs are mostly used in the call of Send_Message(). The use of two types of PIDs, virtual PIDs and physical PIDs, can be made transparent to a user. Two PIDs are of special importance to each uProcess: the uProcess' own PID, and its parent's PID, which can be acquired by system primitives uProcess_Self() and uProcess_Parent(), respectively. In both of cases, the returned PIDs are physical PIDs. If a uProcess needs to know its child uProcess PID, it can request this PID when the call of uProcess_Create() is made. This returned PID is a virtual PID. The parent uProcess then can send messages toward the child uProcess with a virtual PID without worrying about the physical existence of its child uProcess.

From the system's point of view, each message should be sent to the processor indicated by its destination PID. When the message reaches the destination, we check the flag in its destination PCB to see whether it is a virtual PCB or a physical PCB. If it is a virtual PCB, the message will be either forwarded to its physical PID, if available, or put into a pending queue in the virtual PCB. The major steps in handling of messages are given in Figure 5.

As a simple example shown in Figure 6, uProcess A is running on processor 1. The uProcess A creates uProcess B and retains its PID in variable *pid_b*. Then, uProcess B is scheduled on processor 2. The uProcess B obtains the PID of its parent and stores it in variable *pid_a*. Now, both uProcess A and uProcess B can send messages to each other by using the call of Send_Message(). However, the actual message routes are different. Since uProcess A sends a message towards a virtual PID, the message is first routed to the virtual PCB of uProcess B on processor 1. From there it is forwarded to the physical PCB on processor 2. The uProcess B uses a physical PID obtained via the call uProcess_Parent() and, therefore, the message is sent to the physical PCB of uProcess A directly.

3.3. Scheduling Support

The scheduling module decides where a uProcess message will go based on the current system status. We integrated into this uThread package various scheduling algorithms including the Random Allocation algorithm, the Gradient Diffusion algorithm, the Adaptive Contracting Within Neighborhood (ACWN) algorithm, and the Runtime Incremental Parallel Scheduling algorithm.

A random allocation strategy dictates that each processor, when it generates a new uProcess,

- MESSAGE GENERATING: when the Send_Message() call is made
    extract the destination processor number from its destination PID
    if the current processor is the destination processor
        apply Message Routing function described below
    else send the message to its destination

- MESSAGE ROUTING: when a uThread message arrives at processor
    extract the address of PCB from its destination PID
    find out the uProcess' PCB where the message is sent
    check the PCB to see whether it is a virtual or physical PCB
    if it is a virtual PCB
        if the physical PID is available in the virtual PCB
            resend the message to its physical PID
        else put the message in the pending queue of the virtual PCB
    else put the message into the queue waiting for processing

- MESSAGE PROCESSING: when a uThread message reaches the head of the message queue
    extract the address of PCB from its destination PID
    find out the uProcess' PCB where the message is sent
    extract SDA from the PCB
    execute the uThread code with the data and SDA passed to it

Figure 5: Algorithms for Message Handling.

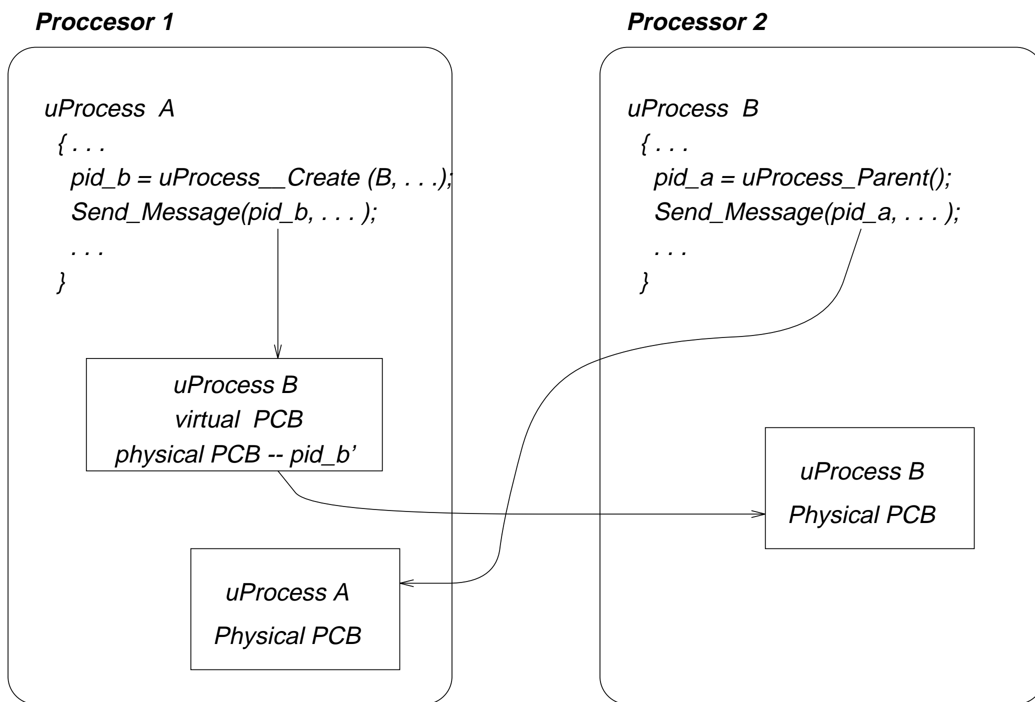

Figure 6: Two uProcesses send messages to each other.

should send it to a randomly chosen processor [4, 9]. One advantage of this strategy is its simplicity of implementation. However, the lack of locality leads to large overhead and communication traffic. Most messages between uProcesses have to cross processor boundaries. This leads to a higher communication load on large systems. Since the bandwidth consumed by a long-distance message is certainly larger, the system is more likely to be communication bound compared to a system using other load balancing strategies that encourage locality.

In the Gradient Diffusion model [12], instead of trying to allocate a newly generated uProcess to another processor, the uProcess is queued at the generating processor and waits for some processor to request it. The scheduling module periodically updates *proximity* on each processor. The proximity of a processor represents an estimate of the shortest distance to an idle processor. An idle processor has a proximity of zero. For all other processors, the proximity is one more than the smallest proximity among the nearest neighbors. When the load is unbalanced, the processor sends a uProcess from its local queue to the neighbor with the least proximity.

ACWN is an adaptive scheduling algorithm [20]. In this algorithm, each processor calculates its own load function and adjacent processors exchange their load information periodically. Thus, each processor maintains information about load on all its nearest neighbors. Depending on how heavily its neighbors are loaded, a newly created uProcess contractes several hops immediately, traveling along the steepest load gradient to a local minimum. If all neighbors are heavily loaded, a processor accumulates uProcesses without sending them out to encourage good locality.

The Runtime Incremental Parallel Scheduling is an alternative strategy to the commonly used dynamic scheduling [21]. In this algorithm, the system scheduling activity alternates with the underlying computation work during *runtime*. The uProcesses are *incrementally* generated and scheduled in *parallel*. The scheduling algorithm includes two major components: incremental scheduling and parallel scheduling. The incremental scheduling policy decides when to transfer a computation phase to a system phase and which uProcesses are selected for scheduling. The parallel scheduling algorithm is applied in the system phase to collect system load information and to balance the load. The Runtime Incremental Parallel Scheduling utilizes advanced parallel scheduling techniques to produce a low-overhead, high-quality load balancing.

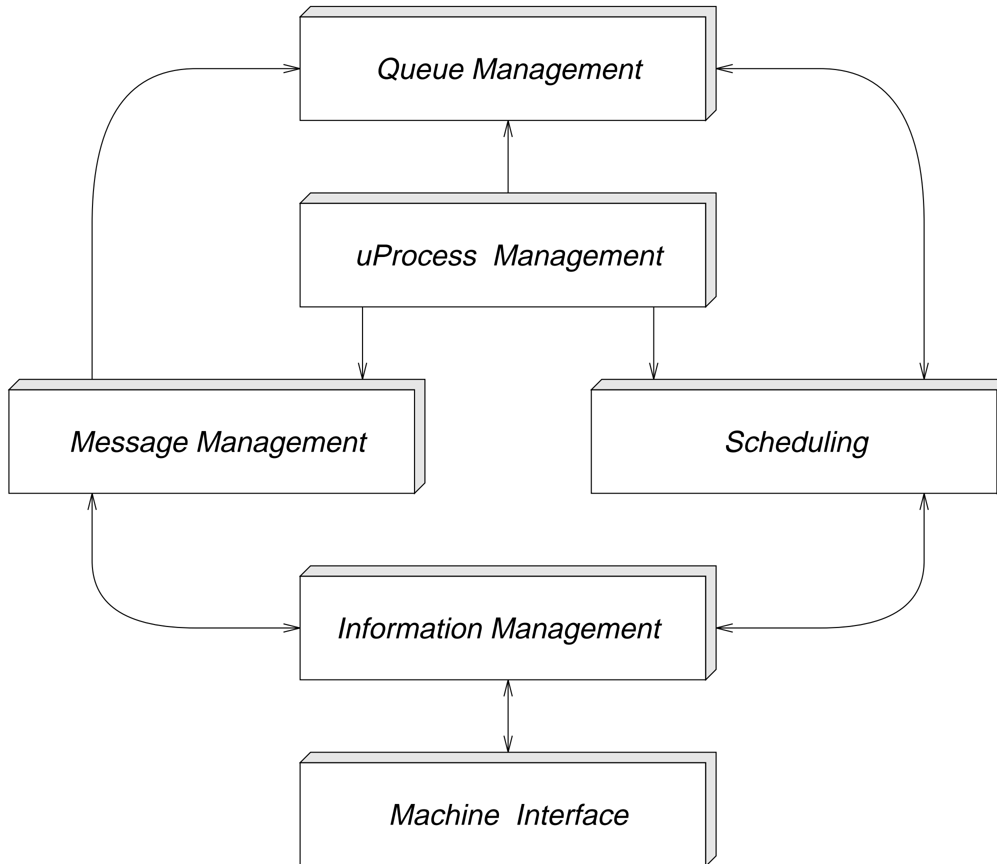

Figure 7: The uThread Package Implementation in Modules

Details of these algorithms and some comparisons can be found in [20, 21].

3.4. Current Implementation

The uThread package consists of six modules as shown in Figure 7: machine interface module, information management module, queue management module, uProcess management module, scheduling module, and message management module. All modules except the machine interface module are machine-independent. Porting the uThread package to a new system is thus a simple matter of defining the machine interface module, and linking it with other modules.

We have implemented the uThread package on Intel iPSC/860, Touchstone Delta, NCUBE, and TMC CM-5. A uProcess creation costs about one hundred instructions and a uThread creation costs less than ten instructions. In comparison, a process creation costs a few tens of

thousands of instructions and a thread creation costs more than three hundreds instructions [2].

4. Related Work

The work presented here is similar to the user-level thread packages, including C Threads and User-level Threads with Continuations at Carnegie Mellon University [7, 8], Sun Microsystem's multi-thread [17, 23], Scheduler Activations at the University of Washington [3], and First-Class threads at the University of Rochester [13]. Most of these works are for general-purpose threads in uniprocessor or shared-memory multiprocessor environments. The major problem these works tackle is the coordination between the kernel-level processes and the user-level threads. Since uThreads are used in the context of massive parallelization of a single application, the elimination of all kernel-level intervention as well as minimization of user-level overhead becomes a central issue. The major difference between uThread and other thread packages is that uThread eliminates context switching overhead by using the RC thread concept. Without supporting multiprocessing, we have only one kernel-level process running at each processor. There is no global shared memory space supported among the processors, and therefore, no mutual-exclusion primitives are needed. All events are synchronized by message passing. In addition, current available distributed memory machines do not support virtual memory. The page-fault blocking can be avoided, too. The work by Bershad, *et al.* on *lightweight RPC* [5] also represents efforts to reduce unnecessary cost in the domain of remote procedure calls. More general issues about processes and threads have been discussed in [24, 22, 2].

The uThread package is based on a message-driven model combined with partially shared local data. Research projects on message-driven models include the J-Machine at MIT [15] and Monsoon at Berkeley [16], in which a messages contains the name of a handler and data. When the message reaches the head of a scheduling queue, the handler is executed with the data as arguments. The execution can be synchronized and suspended. Another message-driven model, the *active messages*, carries the address of a user-level instruction sequence to extract the message from the network and integrate it into the on-going computation [25]. The active messages are used to overlap communication and computation, reducing unnecessary communication overhead. Both active messages and uThreads are message-driven, but, in the active messages, the action taken at arrival of a message is to preprocess the message, whereas, in the uThreads, the action

taken for each message is to perform the primary computation.

Task scheduling can be classified into two categories: static and dynamic. A static scheduling algorithm can schedule applications with static, regular structures [27, 28]. Dynamic scheduling can be applied to general problems with nonuniformly structured problems. The uThread package is especially attractive as a vehicle in dynamic scheduling and load balancing. Numerous algorithms have been studied on dynamic scheduling and load balancing [9, 10, 12, 6]. Many of them are designed for a distributed system instead of a massively parallel environment. A good experimental work, implemented with several dynamic load balancing strategies, is presented in [26].

Implementation of the uThread package is entirely in user space to provide us with great convenience. Early works in a similar context include *MOOSE* at Caltech [19, 11]. Recent development of the microkernel approach to modern operating systems [1, 14, 18] particularly encourages customized thread packages to suit needs of different application and languages.

5. Conclusion

We argue for our approach in terms of both flexibility and performance. The design of the uThread package was heavily influenced by the need to support parallel programming with nonuniform features. In particular, we have attempted to ensure that the overhead introduced by control and manipulation of parallelism can be nearly minimized. The design was also influenced by the need to provide effective scheduling support. The key features of our approach are (1) we simplify the notation of processes when they are employed in exploit of parallelism within a single application; (2) the package is designed for distributed memory environment and supports suitable scheduling strategies; and (3) the package is implemented in user space, portable to most commercially available distributed memory parallel computers and easily extended to newly available parallel machines.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer 1986 USENIX Conf.*, pages

93–112, 1986.

[2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings Publishing Company, Inc., 1994.

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

[4] W. C. Athas. *Fine Grain Concurrent Computations.* PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. on Computer Systems*, 8(1):37–55, February 1990.

[6] S. J. Chapin. *Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems.* PhD thesis, Purdue University, 1993.

[7] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, February 1988.

[8] R. Dean. Using continuations to build a user-level threads library. In *Proc. of the third USENIX Mach Conf.*, April 1993.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, SE-12(5):662–674, May 1986.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Eval.*, 6(1):53–68, March 1986.

[11] J. Koller. The MOOSE II operating system and dynamic loaad balancing. In *The Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 599–602, January 1988.

[12] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Software Engineering*, 13:32–38, January 1987.

[13] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *ACM Symposium on the Principle of Operating System, in ACM Operating System Review*, volume 25, pages 483–490, October 1991.

[14] S. J. Mullender, G. Rossum, A. S. Tanenbaum, R. Renesse, and H. Staveren. Amoeba – a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[15] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: An architecture evaluation. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture*, pages 224–235, May 1993.

[16] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proc. of the 17 Annual Int'l Symp. on Computer Architecture*, May 1990.

[17] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *Proc. of the Winter 1991 USENIX Conf.*, pages 65–79, January 1991.

[18] M. Rozier. Chorus. In *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992.

[19] J. Salmon, S. Callahan, J. Flower, and A. Kolawa. MOOSE: A multi-tasking operating system for hypercubes. In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, pages 391–396, January 1988.

[20] W. Shu. Dynamic scheduling of medium-grained processes on distributed memory computers. In *Proc. of 27th Hawaii Int'l Conf. on System Sciences*, pages 435–444, January 1994.

[21] W. Shu. Runtime incremental parallel scheduling (RIPS) on distributed memory computers. Technical Report 94-25, Dept. of Computer Science, State University of New York at Buffalo, June 1994.

[22] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill, Inc., 1994.

[23] D. Stein and D. Shah. Implementing lightweight threads. In *Proc. of the Summer 1992 USENIX Conf.*, pages 1–9, June 1992.

[24] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Inc., 1992.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19 Annual Int'l Symp. on Computer Architecture*, pages 256–266, May 1992.

[26] Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel and Distributed System*, 9(4):979–993, September 1993.

[27] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[28] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message-passing multiprocessors. *The 6th ACM Int'l Conf. on Supercomputing*, July 1992.