1995

# A Multithreaded Message Passing Environment for ATM LAN/WAN

Rajesh Yadav
*Syracuse University*, ryadav@npac.syr.edu

Rajashekar Reddy
*Syracuse University*, kraja@npac.syr.edu

Salim Hariri
*Syracuse University*

Geoffrey C. Fox
*Syracuse University*

# A Multithreaded Message Passing Environment for ATM LAN/WAN [1]

Rajesh Yadav, Rajashekar Reddy, Salim Hariri, and Geoffrey Fox

Northeast Parallel Architectures Center (NPAC)
Syracuse University
Syracuse, NY 13244
{ryadav, kraja, hariri}@npac.syr.edu

**Abstract**

*Large scale High Performance Computing and Communication (HPCC) applications (e.g. Video-on-Demand, and HPDC) would require storage and processing capabilities which are beyond existing single computer systems. The current advances in networking technology (e.g. ATM) have made high performance network computing an attractive computing environment for such applications. However, using only high speed network is not sufficient to achieve high performance distributed computing environment unless some hardware and software problems have been resolved. These problems include the limited communication bandwidth available to the application, high overhead associated with context switching, redundant data copying during protocol processing and lack of support to overlap computation and communication at application level. In this paper, we propose a Multithreaded Message Passing system for Parallel/Distributed processing, that we refer to as NYNET Communication System (NCS). NCS, being developed for NYNET (ATM wide area network testbed), is built on top of an ATM Application Programmer Interface (API). The Multithreaded environment allows applications to overlap computations and communications and provide a modular approach to support efficiently HPDC applications with different quality of service (QOS) requirements.*

[1] This research is funded by Rome Laboratory (Contract # F30602-94-C-0165), Rome, NY

# 1 Introduction

Large scale High Performance Computing and Communication (HPCC) applications (e.g. Video-on-Demand and HPDC) would require storage and processing capabilities which are beyond existing single high performance computer systems. Current advances in processor technology, and advent of high speed networking technology such as ATM have made high performance network computing an attractive computing environment for such applications. Aggregate power of workstation clusters (Alpha cluster for example) is comparable to that of a supercomputer [10]. Compared to massively parallel computers, network computing is typically less expensive and more flexible.

Another reason for recent rapid growth in network computing area is the availability of Parallel/Distributed tools that simplify process management, inter-processor communication and program debugging in distributed computing environment. However, these advances cannot be fully exploited unless some hardware and software problems are resolved. These problems can be attributed to the high cost of operating system calls, context switching and the use of inefficient communication protocols (e.g TCP/IP). The complexity and the inefficiency of these communication protocols can be justified in 1970s where the networks were slow (operating in Kbps range) and not very reliable while the processing power was relatively three orders of magnitude higher than the network speed.

With advent of high speed networks operating, at Mbps and Gbps, new methods are needed to process protocols efficiently. Reducing the communication latency has been active research area in parallel processing field. Most of the techniques proposed are based on using active messages [2], reducing operating system overhead [1], and overlap computation with communication using Multithreading [7]. In distributed computing, most of the research focused on developing new communication protocols (XTP, FLIP) [11], streamlining existing ones (by merging several layers into one), or building intelligent hardware interface to off-load host from protocol processing.

The main objective of the research presented in this paper is to implement NYNET Communication System (NCS) based on these techniques (e.g. multithreading, reduce data copying and operating system overhead and parallel data transfer) that have been proven to be successful in parallel as well as distributed computing. NCS uses multithreading capability to provide efficient techniques to overlap computation and communication. Furthermore, multithreaded message passing represents an interesting distributed programming paradigm

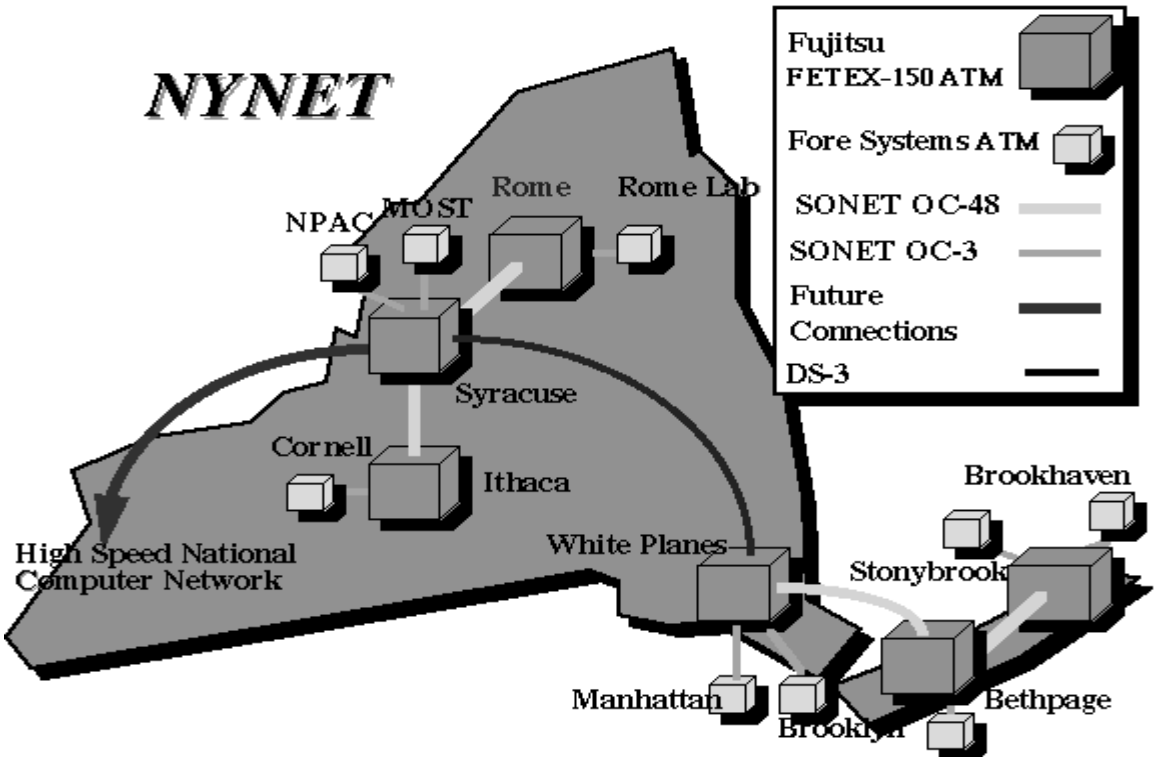



Figure 1: The NYNET ATM Network Testbed

to support efficiently a wide range of Quality-of-Service (QOS) requirements. NCS uses multiple Input/Output buffers to allow parallel data transfer and thus reduces transmission and receiving time. NCS is implemented on top of an ATM API and uses its own read/write trap routines to reduce latency and avoid using inefficient communication protocols (e.g TCP/IP).

The rest of the paper is organized as follows. Section 2 describes the experimental environment used in our benchmarking and outlines the specifications of the hardware and software tool used. Section 3 and 4 describe the design approach and implementation issues of NCS system, respectively. Section 5 presents the performance results using NCS to implement several distributed computing applications. Section 6 summarizes and concludes the paper.

## 2 Experimental Environment

NCS is being developed as part of a larger project involving the development of high performance computing and communication applications for the NYNET (ATM Wide Area

Network) testbed shown in Figure 1. NYNET is a high-speed fiber-optic communications network linking multiple computing, communications, and research facilities in New York State. NYNET ATM testbed uses high speed ATM switches interconnected by fiber-optic SONET (Synchronous Optical Network) links to integrate the parallel computers and supercomputers available at NYNET sites into one virtual computing environment. Most of the wide area portion of the NYNET operates at speed OC 48 (2.4 Gbps) while each site is connected with two OC 3 links (155 Mbps). The upstate to downstate connection is through DS-3 (45 Mbps) link.

In this paper, we report on the performance gain that can be achieved when NCS is used to develop several HPDC applications. These applications include JPEG compression/decompression, Fast Fourier transform (FFT) and parallel matrix multiplication. These applications have been benchmarked on several high performance distributed systems, that are briefly described below.

**SUN/ATM LAN/WAN:** This configuration consists of SUN SPARCstation IPXs interconnected by an ATM LAN using an ATM FORE switch. SUN IPX nodes operate on a approximately 40MHz clock. Host computers are connected to the ATM switch through Fore's SBA-200 SBus adaptors. The SBA-200 has a dedicated Intel i960 processor (running at 25 MHz) to support segmentation and reassemble functions and to manage data transfer between the adaptor and the host computer. The SBA-200 also has special hardware for AAL CRC and special-purpose DMA hardware. 140 Mbps TAXI interface is provided between the workstations and the ATM switch. SUN/ATM WANhas similar characteristics to the SUN/ATM LAN except that the IPXs are now interconnected through the NYNET testbed.

**SUN/Ethernet:** This configuration consists of SUN SPARCstation ELCs interconnected by traditional Ethernet LAN. The ELCs operate at a clock rate of approximately 33 MHz.

## 3 NCS Design Approach

Our approach to implement the NYNET Communication System is based on the following strategies.

- **Simplicity:** Most message-passing systems have been built on top of traditional communication protocols (e.g., TCP/IP and UDP/IP) which were developed to run in

adverse conditions and unreliable networks. But the advent of reliable, high speed networks makes most of the functions provided by these protocols unnecessary. NCS avoids using complex communication protocols and instead uses the ATM API to implement the required communication services.

- **Parallel Data Transfer:** To reduce the data transfer time for send and receive operations, NCS uses multiple input/output buffers as shown in Figure 2. In this scheme, NCS copies data to be sent to the first output buffer and then signals the network interface. The network interface starts transferring the data in the first buffer while NCS is filling the second output buffer. Similar technique is used to reduce the receiving time by using the multiple input buffers.

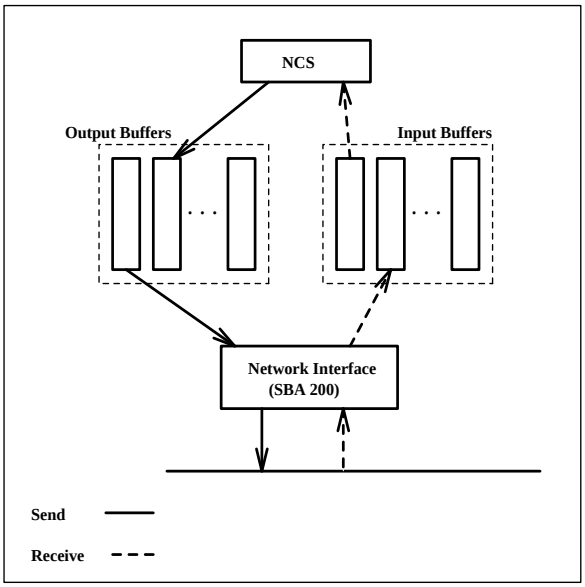



Figure 2: Concurrent Data Transfers

- **Reduce number of data accesses:** The path taken by data as it passes a stack of communication layers based on Unix Sockets and TCP/IP is shown in Figure 3(a). During data transmission, the application writes data into the application buffer and then invokes a system call to send the data. The socket layer copies the application buffer into a socket buffer in the kernel space. The transport layer (TCP) reads the data in the kernel buffer and modifies it according to the TCP protocol functions. Then, the data is copied out to the network interface. Consequently, the memory bus system is accessed five times for each word of transmitted data. Figure 3(b) shows the data path using NCS system. The application writes the data into the application buffer,

NCS copies the data in the application buffer into the kernel space. System calls are not required because the kernel buffers are made accessible to NCS by mapping these buffers into the NCS address space. Then kernel transfers data from these buffers to the network interface. In this scheme, system bus is accessed only three times and thus reduces the data transfer time. Similar approach is used to receive data from the network.

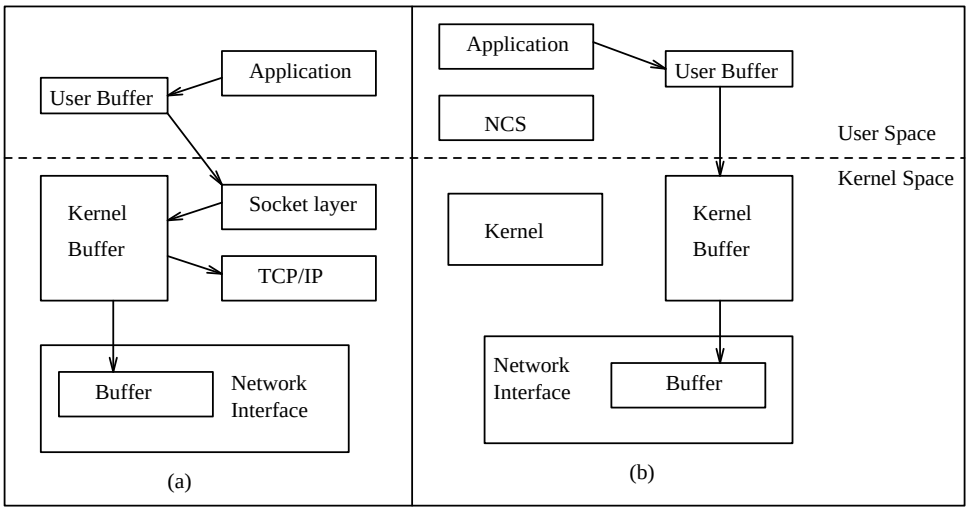
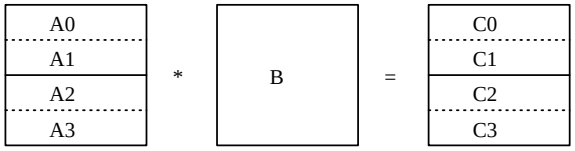



Figure 3: Datapath during Communication

- **Overlap communication and computation:** Overlapping computation and communication is an important feature in network-based computing. In wide area network (WAN) based distributed computing, the propagation delay (limited by the speed of light) is several orders of magnitude greater than the time it takes to actually transmit the data [5]. For example, to transmit one Kbyte file across U.S at 1Gbps transmission rate, it takes only 8 microseconds. However, the time it takes for the first bit to arrive at its destination (propagation delay) is 15 milliseconds. Consequently, the transmission time of this file is insignificant when compared to the propagation delay which cannot be avoided. The only viable approach to reduce the impact of propagation delay is to modify the structure of computations such that we can overlap them with communications. NCS adopts multithread (multiple threads per node) programming paradigm to achieve the desired overlapping and thus reduce the propagation delay impact on HPDC applications. For example, Figure 4 shows how multithread message passing approach can reduce the overall execution time of matrix multiplication. Let us assume

that two processors (with one process on each processor) are involved in the computation. *Process_1* is responsible for calculating C0 and C1 (see Figure 4) and *process_2* is responsible for calculating C2 and C3. If there are no threads *process_2* has to wait till both A2 and A3 are received before it can start its computation. But if there are two threads per process, then *thread_0* of *process_2* can start computing C2 as soon as it receives A2 while *thread_1* is waiting to receive A3. As can be seen from the figure this overlapping reduces the overall execution time.



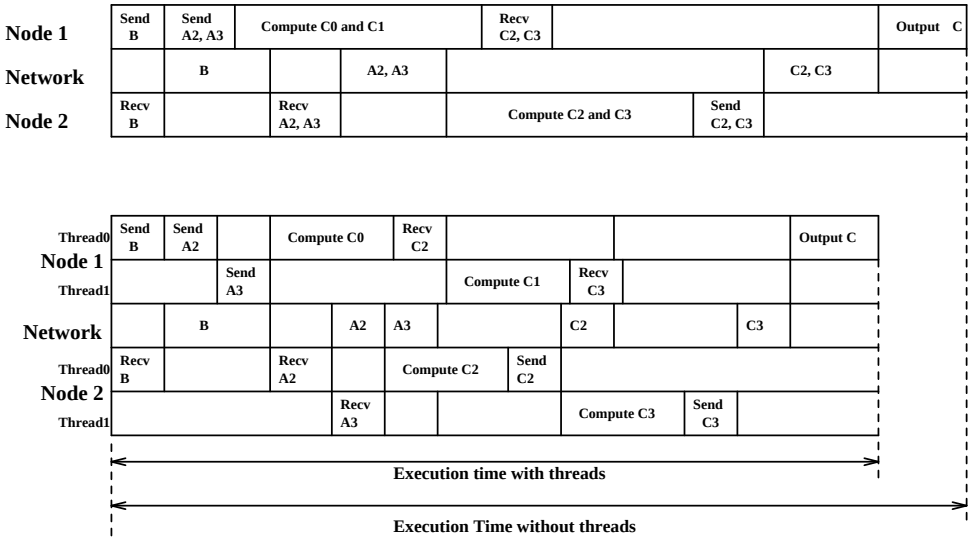




Figure 4: Overlap of Computation and Communication

- **Modular:** Since the multithread approach is modular, it allows us to support a wide range of HPCC Quality of Service (QOS) requirements. Figure 5 shows two applications, Video-On-Demand (VOD) application and a parallel and distributed application (PDA), each of them has a multiple number of compute threads. Their requirements of QOS (e.g flow control) would be different. As shown in the figure, NCS provides different flow control mechanisms such that the one that best suites a given application can be invoked dynamically at runtime.

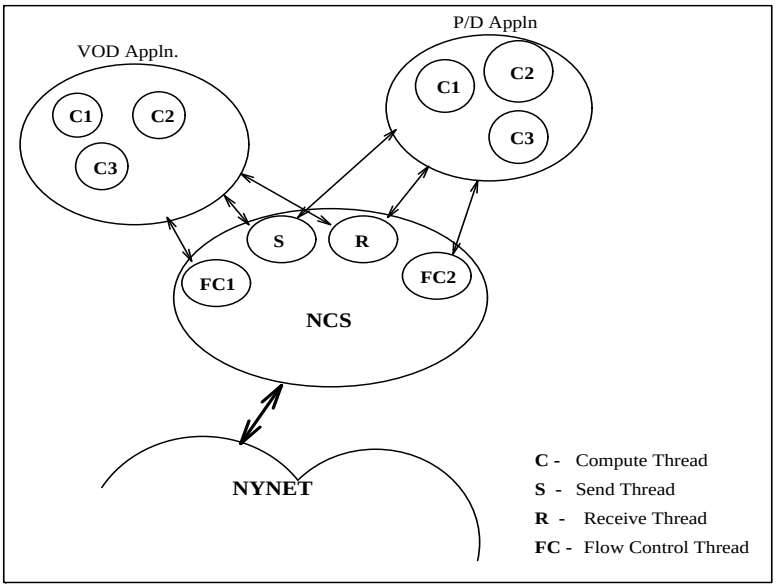



Figure 5: Application QOS requirements are different.

We do believe that it is not possible to build a communication system that can support efficiently all applications. Also there are applications where interoperability requirement is more important than the performance. On the other hand, in real-time parallel and distributed applications performance is essential. NCS provides a framework to address these two conflicting requirements by supporting two classes of applications *viz.*, Normal Speed Mode (NSM) and High Speed Mode (HSM). Figure 6 shows the two tier architecture of NCS. The NSM emphasizes interoperability and uses traditional communication systems (*e.g.* TCP/IP), whereas the HSM uses NCS or other message passing tools ported to NCS, which in turn is built on ATM API. The message passing filters shown in the figure allow p4, PVM and other message passing tools' primitives to be mapped to NCS primitives.

### 3.1 NCS services

Parallel and Distributed Computing tools can be broadly characterized in terms of the following classes [6].

- Point-to-Point Communication: *send* and *receive* are the basic message passing primitives that provide this type of interprocess communication.
- Group Communication: This involves communication among multiple senders and receivers. These are further divided into three classes based on the number of senders and receivers *viz. 1-to-many*, *many-to-1*, and *many-to-many*.

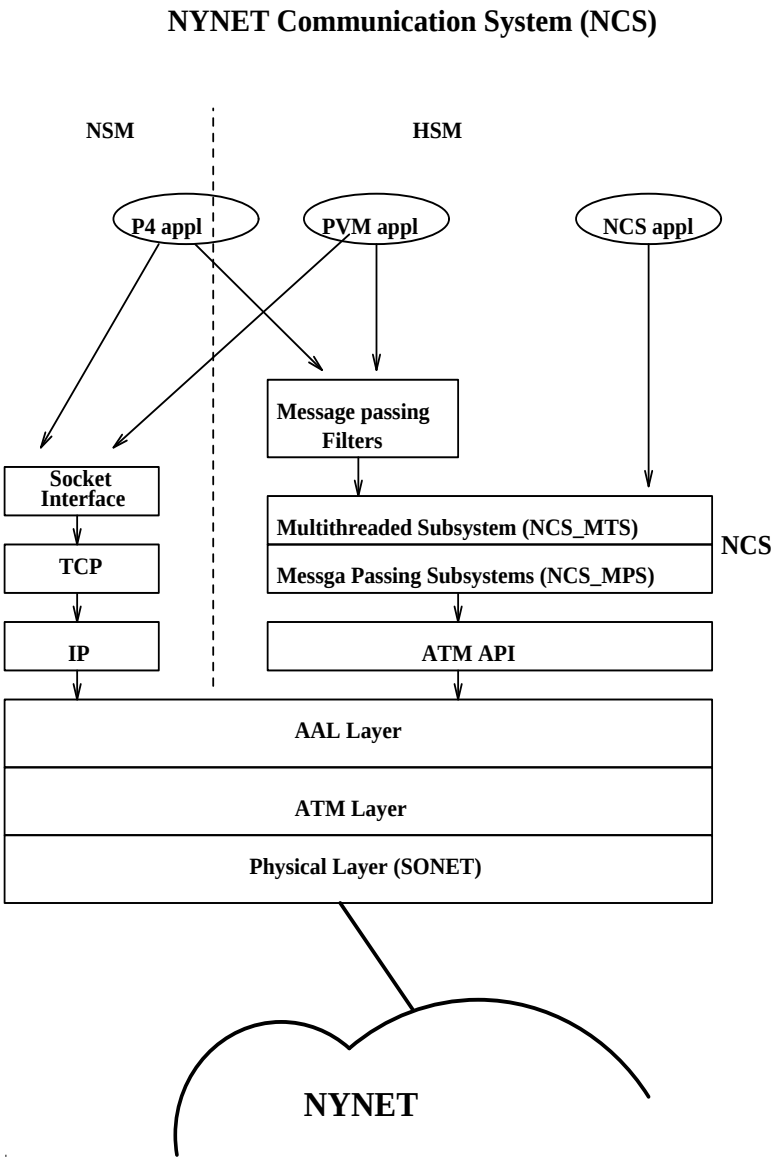



Figure 6: NYNET Communication System

**NCS_send**(int *from_thread*, int *from_process*,
int *to_thread*, int *to_process*,
char **data*, int *size*)
**NCS_recv**(int *from_thread*, int *from_process*,
int *to_thread*, int *to_process*,
char ***data*, int **size*)
**NCS_bcast**(int *from_thread*, int *from_process*,
identifier **list*,
char **data*, int *size*)
where
*from_thread* and *from_process* are the sending thread's and sending process's identifiers.
*to_thread* and *to_process* are the receiving thread's and receiving process's identifiers.
*list* in **NCS_bcast** gives the list of thread and process identifiers to which the data is to be sent.
*data* is pointer to the data to be sent or received.
*size* if the size of the data to be sent or received.

Figure 7: Syntax of NCS primitives

- Synchronization: The primitives in this class are used for synchronization among different processes in the distributed computing environment (*e.g. barrier*, *wait*, *signal*).
- Exception Handling: Exception Handling is more difficult for distributed applications. A few software tools provide functions that handle exceptions.

NCS supports the above classes of functions. Figure 7 gives the syntax of some of the primitives supported by NCS.

# 4 Implementation Issues

Figure 8 shows the main components of the NYNET Communication System and how they are interconnected. The programmer develops HPDC application using multiple compute threads, NCS *send*, *receive*, and *flow control* threads. The compute threads do the actual computation and use NCS calls *NCS_send()* and *NCS_recv()* for communication. These functions wake up the *send* and *receive* threads respectively and block the calling thread. The *send* and *receive* threads do the actual data transfer and when they are done, they wake

up the corresponding compute threads. Also we use multiple input and output buffers to overlap the data transfer between the kernel buffers and the network interface. They are mapped to address space of the NCS system so that NCS has direct access to these buffers.

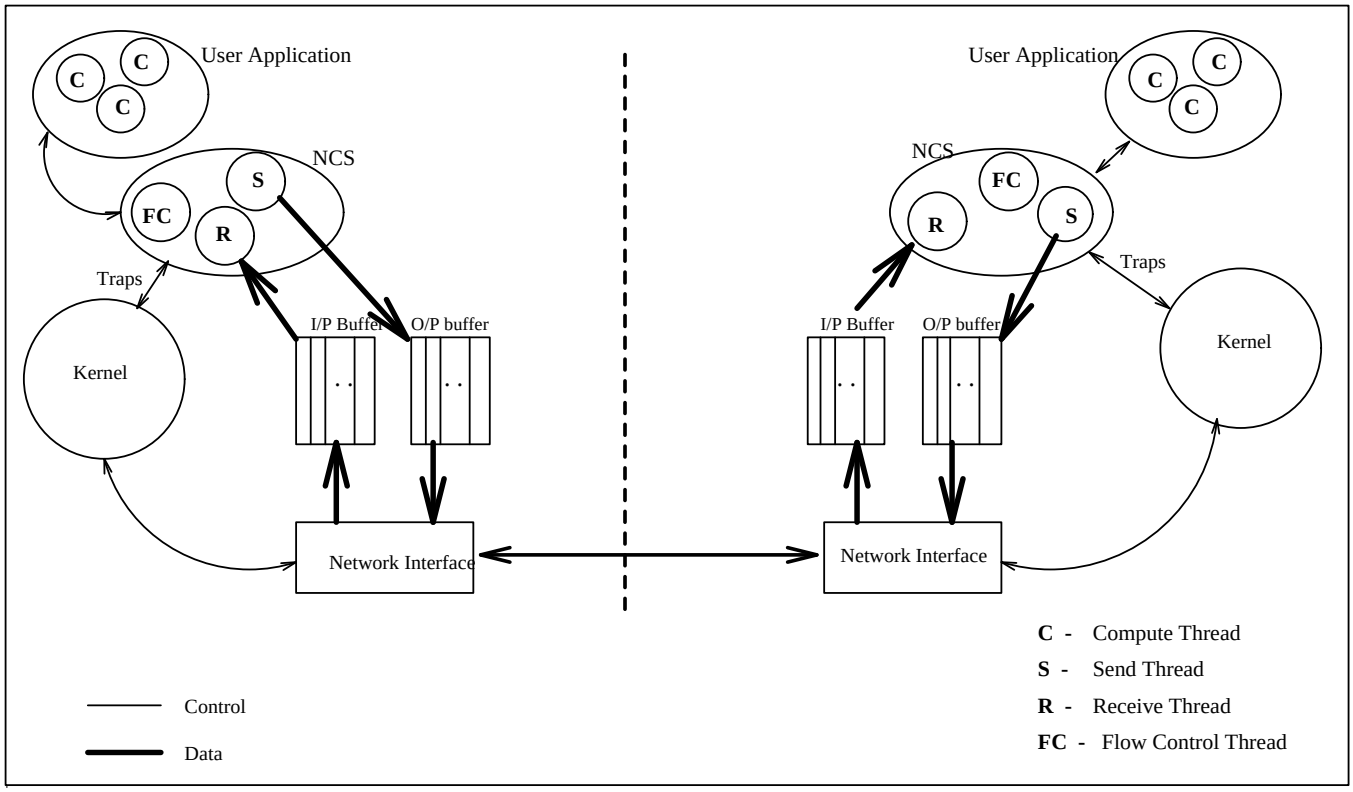



Figure 8: NYNET Communication System Components

The NYNET Communication System can be viewed as two main subsystems.

1. NCS MultiThread Subsystem (NCS_MTS) which provides all thread related services.
2. NCS Message Passing Subsystem (NCS_MPS) which provides the communication services.

In what follows we describe our approach to implement each of these subsystem.

### 4.1 NCS_MTS Implementation

NCS_MTS is built on top of QuickThreads toolkit that has been developed at University of Washington [4]. QuickThreads is a user-space thread toolkit, that is the operating system of the host has no information about the number of the threads running and their states. Threads are realized within a conventional process and thread management is done by the

run-time system. Also, the data structures for threads are maintained in shared memory. QuickThreads is not a stand-alone threads package, rather it is used to build user-level thread packages. It only provides the capability for thread initialization and context switching. We have added scheduling and synchronization capabilities to Quick thread that are needed for implementing NCS_MTS. Also, NCS_MTS can support several scheduling and synchronization techniques. In NCS_MTS, there are N priority levels (current implementation has N = 16) and within each priority level, round robin scheduling scheme is used. We implemented this scheduling mechanism using doubly linked lists (see Figure 9).

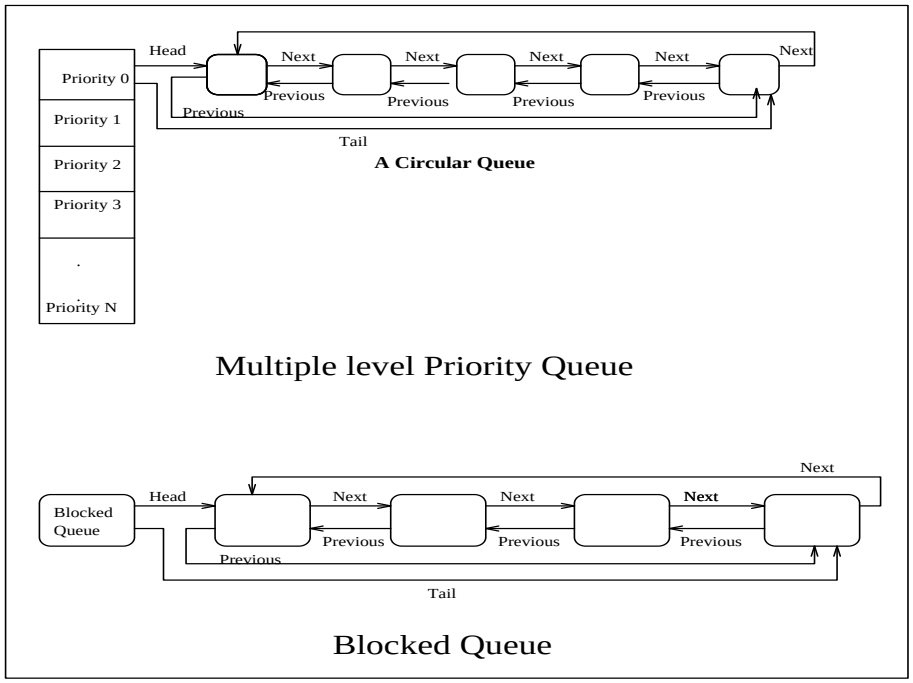



Figure 9: Data-structures for Queues

In NCS_MTS a thread can be in one of three states: *blocked, runnable or running.* Blocking can be viewed as the mechanism that synchronizes a thread with some event (e.g., waiting to receive a message). We implemented blocked queue by doubly linked list to speed up search operation during unblocking of threads. A thread is unblocked when the event it is waiting for, is completed and is placed into runnable queue according to its priority level.

In our implementation we have two classes of threads : System threads and User threads. System threads include *send*, *receive*, *flow control* and *error control* threads. These threads are created during thread environment initialization (*NCS_init(flow, error)*). *NCS_init* takes two arguments which allow the programmer to select the flow and error control threads for

- Main program:
  **Begin**
  - *NCS_init(flow, error)* /* Initialize environment and create system threads */
  - tid1 = *NCS_t_crate(Thread1, arg, priority)*
    /* create computation thread 'Thread1'*/
  - tid2 = *NCS_t_crate(Thread2, arg priority)*
    /* create computation thread 'Thread2'*/
    .
    .
  - tidn = *NCS_t_crate(ThreadN, arg priority)*
    /* create computation thread 'ThreadN'*/
  - *NCS_start()* /* start running the threads */

  **End**

Figure 10: Generic model for application programs

his/her application. If no argument is provided then default flow and error control threads are used. User threads include the computation threads and are created by the application itself (using *NCS_t_create()*). Figure 10 shows a general model on how NCS can be used to develop parallel/distributed applications.

In our current implementation of NCS_MTS, we have only implemented *send* and *receive* system threads and uses the flow and error control provided by p4 [8].

### 4.2 NCS_MPS Implementation

We have considered two approaches for the implementation of NCS_MPS. One approach is based on integrating an existing message passing software tool such as p4 [8] with the NCS_MTS subsystem. The main objective of this approach is to demonstrate that multi-threaded message passing programming paradigm is a viable approach to develop HPDC applications. Figure 11 shows the integration approach of NCS_MTS with p4. In this case, application programs are written using NCS_MTS and underlying message passing layer (e.g., p4) is hidden from programmer. We have developed basic NCS non-blocking communication primitives *NCS_send()* and *NCS_recv()* using calls provided by p4, *viz. p4_messages_available(), p4_send() and p4_recv()*. Non-blocking in the sense that these calls block only the thread which calls them but do not block the whole process. This allows other threads to run

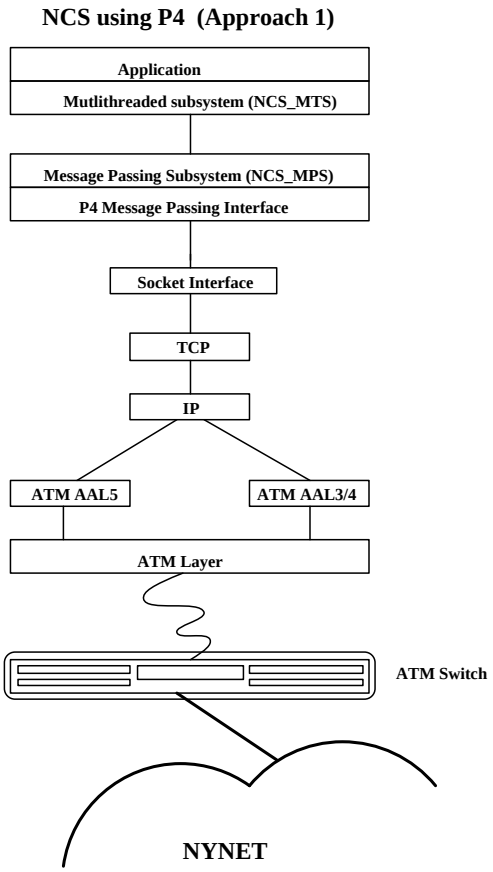



Figure 11: NCS_MPS implementation using p4

and do useful work and thus overlap computations and communications. *NCS_recv* is called when it is required to receive a message from another thread either remote or local. This function wakes up the receive thread and blocks the calling thread. This blocked thread is unblocked by the *receive* thread when it receives the required message. Meanwhile, other threads can continue their computations. We have seen a significant performance gain by using multithreaded message passing programming paradigm to develop high performance distributed computing applications as will be discussed later in section 5. We are currently investigating the performance of implementing NCS using other message passing tool such as PVM [12] and MPI [13].

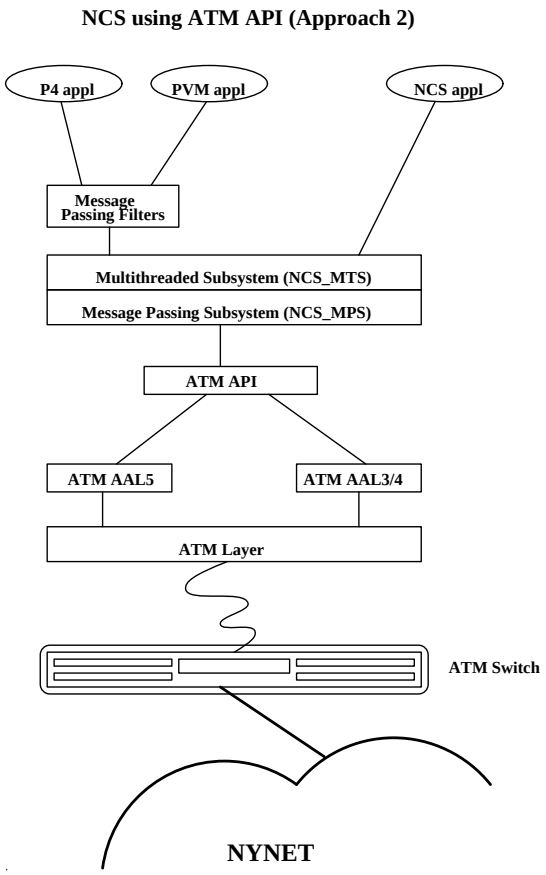



Figure 12: NCS_MPS implementation using ATM API

Our second approach avoids using traditional communication protocols (e.g., TCP/IP) and uses modified ATM API. Figure 12 shows this implementation approach of the NCS_MPS. The second approach does not change the implementation of NCS applications that have been developed based on NCS_MTS/p4 implementation.

In this second implementation, we reduce the overhead associated with data-copying

operations by allowing NCS to access the kernel buffers without context switching. This would be done by using UNIX system call *mmap()* to map the kernel buffer into user space. Furthermore, we use traps to transfer control between NCS and UNIX kernel, and to read and write from network interface. The use of traps has been shown to be more efficient than using UNIX read/write system calls [1]. We will also develop the message passing filters for the commonly used message passing tools (e.g., p4, PVM, MPI) so that any parallel/distributed application written using these tools can be ported to NCS without any change.

# 5 Benchmark Results

In this section we evaluate the performance of the first implementation of NCS using three applications. The second implementation is not fully operational when this paper is written. We also compare the performance of the NCS implementation of these applications against the performance of using only message passing tool p4 [8].

## 5.1 Matrix Multiplication

We have used a simple distributed matrix multiplication algorithm since our intent is to compare the performance of NCS implementation with p4. Given A and B matrices, the problem is to compute C = A*B. We have used the host-node programming model. The *host_process* sends the whole B matrix to all the *node_process* and distributes the rows of A matrix equally among the nodes. Each of the node processes then calculates its portion of the C matrix and sends the result to the *host_process*. Figure 13 shows an implementation of this algorithm in p4.

Figure 14 gives an implementation of this algorithm using NCS, where we assume there are two threads per process. In this implementation *thread_0* of *host_process* communicates with *thread_0* of node_processes and calculates the one half of the C matrix. Similarly, *thread_1* of *host_process* calculates the other half of C matrix. Notice that B matrix is sent to a particular node only once, since all the threads share the same address space on a particular node.

Table 1 shows the performance that can be gained by using NCS_MTS/p4. The execution time for one node is approximately equal for both p4 and NCS_MTS/p4 which is expected. The small difference is because NCS_MTS/p4 implementation has overhead of maintaining threads. In general, NCS_MTS/p4 implementation outperforms the p4 implementation because of overlapping computations and communications. For examples, for 4 nodes col-

```
main(argc, argv){
    p4_initenv(argc, argv) /* initialize p4 environment */
    p4_create_procgroup() /* create remote processes */
    my_id = p4_get_my_id()
    if(my_id == HOST)
       Call Host_Process()
    else
       Call Node_Process()
}

Host_Process(){
   /* Distribute matrix */
   for i = 1 to num_nodes
       p4_send(DATA, i, B, sizeB);
       p4_send(DATA, i, &A[offset], size);
   /* Wait for results */
   for i = 1 to num_nodes
       type = RESULT;
      from = i;
      p4_recv(&type, &from, &C[offset], &size);
}

Node_Process(){
    /* Receive data from host */
       type = DATA;
       from = HOST;
       p4_recv(&type, &from, &B, &sizeB);
       p4_recv(&type, &from, &A, &size);
    /* Calculate the resultant matrix */
        Call matrix_compute(C)
    /* Send back the resultant matrix */
        p4_send(RESULT, HOST, C, size);
}
```

Figure 13: Matrix Multiplication in p4 Environment

```
main(argc, argv){
    p4_initenv(argc, argv) /* initialize p4 environment */
    p4_create_procgroup() /* create remote processes */
    NCS_init() /* Initialize thread environment */
    tid1 = NCS_t_create((void(*)())Compute_matrix1, arg1, priority)
    tid2 = NCS_t_create((void)(*))Compute_matrix2, arg2 priority)
    NCS_start() /* start running the threads */
}
Host Process
Compute_matrix1(){
   /* Distribute matrix */
       for i = 1 to num_nodes
         NCS_send(THREAD1, HOST, THREAD1, i, B, sizeB);
         NCS_send(THREAD1, HOST, THREAD1, i, A[offset], size);
   /* Wait for results */
      for i = 1 to num_nodes
      NCS_recv(THREAD1, i, THREAD1, HOST, &C[offset], &size);
}
Compute_matrix2(){
   Similar to Compute_matrix1 except that send and receive from
   THREAD2 of nodes instead of THREAD1.
}
Node Process
Compute_matrix1(){
    /* Receive data from host */
       NCS_recv(THREAD1, HOST, THREAD1, my_id, &B, &sizeB);
       NCS_recv(THREAD1, HOST, THREAD1, my_id, &A, &sizeB);
    /* Calculate the resultant matrix */
        Call matrix_compute(C)
    /* Send back the resultant matrix */
        NCS_send(THREAD1, my_id, THREAD1, HOST, C, size);
}
Compute_matrix2(){
   Similar to Compute_matrix1 except that send and receive
   from THREAD2 of host instead of THREAD1.
}
```

Figure 14: Matrix Multiplication in Multithread Message Passing environment

Table 1: Execution times of Matrix Multiplication (seconds)

| Nodes | Ethernet | | | NYNET Testbed | | |
|---|---|---|---|---|---|---|
| | p4 | NCS_MTS/p4 | % Improvement | p4 | NCS_MTS/p4 | % Improvement |
| 1 | 25.77 | 25.85 | - | 24.89 | 25.03 | - |
| 2 | 16.89 | 13.72 | 18.76% | 14.4 | 11.51 | 20.06% |
| 4 | 10.64 | 7.88 | 25.93% | 7.52 | 5.41 | 28.05% |
| 8 | 5.90 | 4.62 | 21.69% | - | - | - |

laborating on a matrix multiplication of size 128x128, the performance gain (difference in execution times over p4 execution time) of using NCS_MTS/p4 versus p4 is 26% on Ethernet and it is around 28% on NYNET testbed.

The execution times associated with NYNET testbed are better because of two reasons: the computers connected to ATM network are faster machines and ATM network operates at a faster speed than Ethernet.

### 5.2 JPEG Compression/Decompression

JPEG (Joint Photographic Experts Group) is emerging as a standard for image compression. JPEG standard aims to be generic and can support a wide variety of applications for continuous-tone images. We have used data parallel programming model to implement a distributed JPEG algorithm on a cluster of workstations. In this implementation half of the computer participate in compression of an image file while the second half reconstruct the compressed image. The image to be compressed is divided into N/2 equal parts (where N denotes the number of processors) by the master process and then shipped to one half of the processors. Each processor performs the sequential JPEG compression algorithm on its portion of the image. After compression the processors send the compressed image to another set of N/2 processors which perform the decompression. Once decompression is done, the results are sent back to the master process which combines them into one image. Consequently, this algorithm involves five stages *viz.* distribution of uncompressed image , compression of the image, transmission of compressed image, decompression of the image, and finally combining the decompressed images.

In multithreaded environment we have two computation threads running on each processor, so if one thread is blocked for communication, the other thread can run and perform useful work. The general communication pattern between threads on different processor is

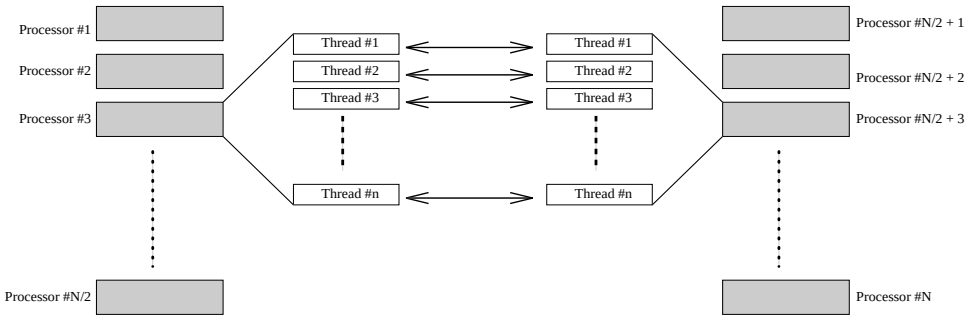



Figure 15: General Communication Pattern in JPEG Compression/Decompression Algorithm

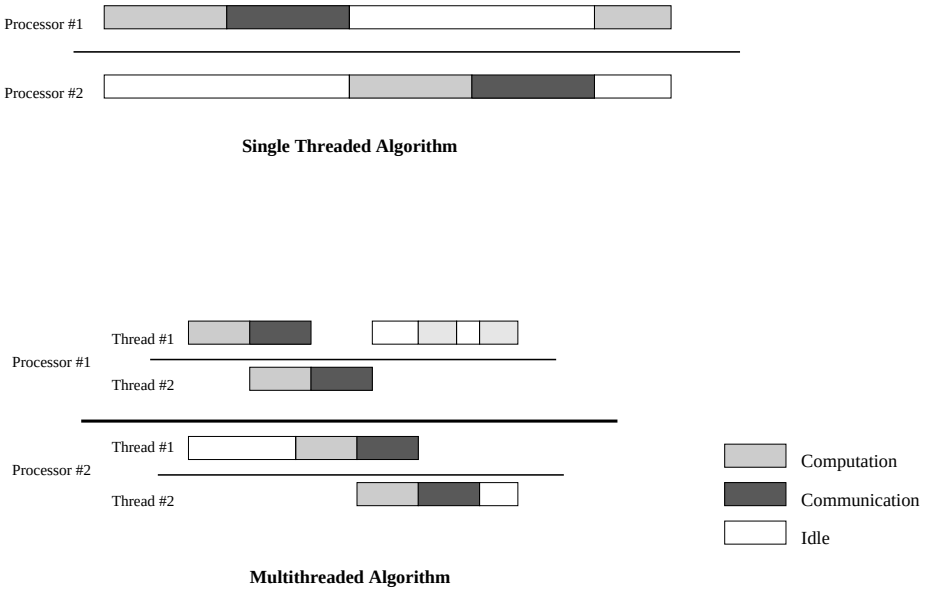



Figure 16: Computation and Communication pattern with two thread/processor

```
main(argc, argv){
    p4_initenv(argc, argv) /* initialize p4 environment */
    p4_create_procgroup() /* create remote processes */
    NCS_init() /* Initialize thread environment */
    tid1 = NCS_t_create((void(*)())Compute_image1, arg1, priority)
    tid2 = NCS_t_create((void)(*))Compute_image2, arg2 priority)
    NCS_start() /* start running the threads */
}
Compute_image1(){
   /* Allocate memory and initialize variables */
       Initialization()
   /* Input the image file */
       while (!EOF) fget(FILE, Image)
   /* Unblock Compute_image2 */
       NCS_unblock(tid2)
   /* Distribute half the image among Thread1 of N/2 processors */
      for (I = 1; I < N/2; I++)
        NCS_send(THREAD1, HOST, THREAD1, I, Image[offset], size)
   /* Compress its portion of image */
      Call jpeg_compress()
   /* Send the compressed image to corresponding processor */
      NCS_send(THREAD1, HOST, THREAD1, N/2, DImage, size)
   /* Combine the received decompressed images */
       for(I= N/2; I < 3N/2; I++)
         NCS_recv(-1, -1, THREAD1, HOST, &Image[offset], &size)
   /* Write decompress image in the output file */
       while(data) fput(FILE, Image)
}
Compute_image2(){
   /* Blocks until image file is read */
       NCS_block()
   /* Distribute remaining half of the image among THREAD2 of N/2 processors */
       For I = 1, 2, .. N/2-1
        NCS_send(THREAD2, HOST, THREAD2, I, Image[offset], size)
   /* Compress its portion of image */
       Call jpeg_compress()
   /* Send the compressed image to corresponding processor */
       NCS_send(THREAD2, HOST, THREAD2, N/2, DImage, size)
}
```

Figure 17: Pseudo code for JPEG Host_Process

```
main(argc, argv){
    p4_initenv(argc, argv) /* initialize p4 environment */
    p4_create_procgroup() /* create remote processes */
    NCS_init() /* Initialize thread environment */
    tid1 = NCS_t_create((void(*)())Compute_image1, arg1, priority)
    tid2 = NCS_t_create((void)(*))Compute_image2, arg2, priority)
    NCS_start() /* start running the threads */
}

Compute_image1(){
    my_id = NCS_get_my_id()
    if (my_id < N/2)
        /* Allocate memory and initialize variables */
        Initialization()
        /* Receive its portion of image from host */
        NCS_recv(THREAD1, HOST, THREAD1, my_id, &Image, &size)
        /* Compress the image */
        Call jpeg_compress()
        /* Send the compressed image to its corresponding processor */
        NCS_send(THREAD1, my_id, THREAD1, N - my_id, DImage, size)
    Else
        /* Receive the compressed image from its corresponding processor */
        NCS_recv(THREAD1, N - my_id, THREAD1, my_id, &DImage, &size)
        /* Decompress the image */
        Call jpeg_decompress()
        /* Send the decompress image to host */
        NCS_send(THREAD1, my_id, THREAD1, HOST, Image, size)
}

Compute_image2(){
    Similar to Thread1 but send and receive from second thread of corresponding
    processors
}
```

Figure 18: Pseudo code for JPEG Node_Process

Table 2: Total execution times (Seconds)

| Nodes | Ethernet | | | NYNET Testbed | | |
|---|---|---|---|---|---|---|
| | p4 | NCS_MPS/p4 | % Improvement | p4 | NCS_MPS/p4 | % Improvement |
| 2 | 10.721 | 9.037 | 15.71% | 6.248 | 4.837 | 22.58% |
| 4 | 15.325 | 8.849 | 42.26% | 10.154 | 4.074 | 59.88% |
| 8 | 17.343 | 6.541 | 62.28% | - | - | - |

shown in Figure 15. Here, N/2 left processors compress the image whereas N/2 right processors decompress the image. Figure 16 shows the state (computation, communication or idle) of each processor during the application execution for two cases:

1. One thread per node (this represents pure message passing implementation).

2. multithreaded implementation with two threads per node.

The Pseudo code implementations of the host and node programs using NCS are shown in Figures 17 and 18, respectively. In this implementation *Thread_1* works on the first half portion of the image while *Thread_2* works on the other half of the image. Each thread sends its compressed image to the corresponding thread on the remote processor for decompression.

Table 2 compares the performance of NCS_MTS/p4 implementation against p4 implementation for 600 Kbyte image on two distributed computing systems. It also shows the percentage improvement of NCS_MTS/p4 implementation over p4 implementation, which is calculated as difference in execution times over p4 execution time. The improvement in performance is consistent with those obtained from the previous application. For example, for 4 nodes working on JPEG compression/decompression performance gain of NCS_MTS/p4 implementation vs p4 implementation is around 42% for Ethernet and 60% on NYNET testbed.

### 5.3 Fast Fourier Transform (FFT)

Given the input sample signal $s(k)$, $k = 0, 1, \ldots, M-1$, computation of FFT gives the output $X(i)$, $i = 0, 1, \ldots, M-1$ such that

$$X(i) = \sum_{k=0}^{M-1} s(k) \cdot W^{ik}$$

where $W = e^{j2\pi/M}$, $j = \sqrt{-1}$ and $M$ = number of sample points.

### 5.3.1 FFT on N workstations using p4

Suppose we have $N$ workstations on the network and $M = N \cdot 2^n$ $(n \geq 1)$, the DIF (Decimation in Frequency) algorithm for FFT can be mapped onto the network of workstations. A case of $M = 8$ and $N = 2$ is shown in Figure 19. Each of the small circles shown in the figure represents a computation which takes two sample inputs and gives two outputs. If sample size is $M$, then there are $M/2$ rows of computation. Each node takes $M/(2 \cdot N)$ rows. The lines crossing the bold lines represent interprocessor communication. There are $\log_2 M$ computation steps and $\log_2 N$ communication steps.

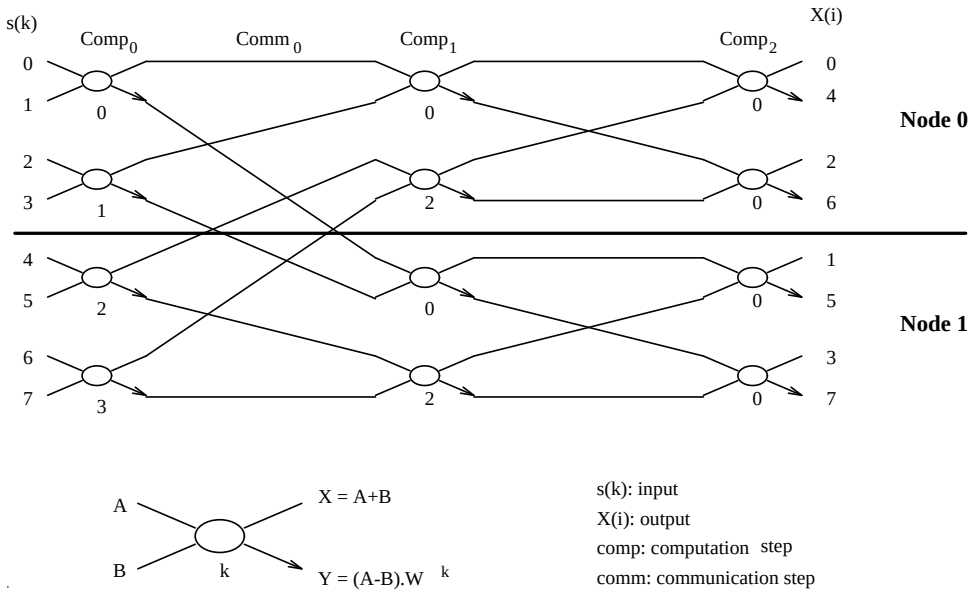



Figure 19: DIF FFT in p4 environment

The algorithm has been implemented using host-node programming model. The *host_process* distributes the sample inputs to the node processes and collects results from the *node_processes*. All the work is being done by *node_processes* which communicate among themselves during computation.

### 5.3.2 FFT on N workstations using p4 and NCS_MPS

In this case, there are multiple threads per process. We assume that there are two threads per *node_process* and *host_process* has only one thread. Thus if we have $N$ workstations, then there are $2 \cdot N$ threads working on the problem. Figure 20 shows how the computation proceeds using this approach for a case of $M = 8$ and $N = 2$. Each thread gets $M/(4 \cdot N)$

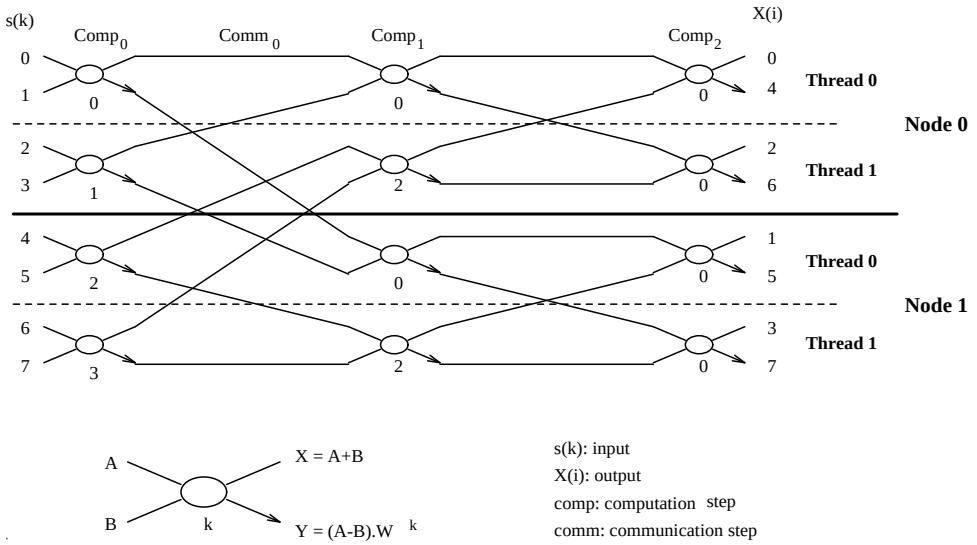



Figure 20: DIF FFT on NCS_MTS/p4 environment

rows of the computation. The bold horizontal lines separates two different nodes and dashed horizontal lines separate different threads.

The host process does the same job as before, it distributes the input points among the different threads equally and collects the results from these threads. The algorithm for two threads of the *node_process* is given in Figure 21. There are $\log_2 M$ computation steps and $\log_2 2 \cdot N$ communication steps. Note that the last communication step is local among threads and does not involve remote communication.

The advantage of this algorithm is that when *thread_0* of a node is waiting to receive data, *thread_1* can continue its computation. This overlap of communication and computation improves the performance as shown in Table 3. In this implementation, we chose the number of sample points, M = 512 and 8 sample sets. As expected, for one node the execution times for both p4 and NCS_MTS/p4 are approximately equal. NCS_MTS/p4 implementation performs better than p4 implementation as the number of nodes is increased. For example, for 4 nodes performance gain of NCS_MTS/p4 implementation vs p4 implementation is 5.7% on Ethernet and 10.66% on NYNET testbed.

# 6 Conclusion

In this paper we presented a multithreaded message passing environment for parallel/distributed computing over ATM LAN/WAN, which is built on top of an ATM Application Programmer

**main(argc, argv){**

p4_initenv(argc, argv) /* initialize p4 environment */
p4_create_procgroup() /* create remote processes */
NCS_init() /* Initialize thread environment */
tid1 = NCS_t_create((void(*)())Compute_fft1, arg1, priority)
tid2 = NCS_t_create((void)(*))Compute_fft2, arg2, priority)
NCS_start() /* start running the threads */

}

**Compute_fft1(){**

NCS_recv(THREAD1, HOST, THREAD1, my_id, &A, &size);
NCS_recv(THREAD1, HOST, THREAD1, my_id, &B, &size);
$thread\_num = 2 \cdot my\_num$; $N2 = 2 \cdot N$;
For $step = 0$ to $\log_2 N2 - 1$
$d = N2 \cdot 2^{-step-1}$; /* communication distance */
$k$= Modulo$(thread_num \cdot M/(2 \cdot N2) \cdot 2^{step},\ M/2)$;
For $i$= 0 to $M/(2 \cdot N2)$ /* computation */
$X_i = A_i + B_i$; $Y_i = (A_i - B_i) \cdot W^k$; $k = k + 2^{step}$;
if Modulo$(thread_num, 2d) < d$ /* communication */
$thread = (thread\_num + d) mod 2$;
$process = (thread\_num + d) div 2$;
NCS_send(THREAD1, my_id, THREAD1 , process, Y, size);
NCS_recv(THREAD1, process, THREAD1, my_id, &B, &size);
$A = X$;
else
$thread = (thread\_num - d) mod 2$;
$process = (thread\_num - d) div 2$;
NCS_send(THREAD1, my_id, THREAD1, node, X, size);
NCS_recv(THREAD1, process, THREAD1, my_id, &A, &size);
$B = Y$;
For $step$= $\log_2 N2$ to $\log_2 M - 1$ /* computation */
$k$ =0;
For $i$= 0 to $M/(2 \cdot N2)$
$X_i = A_i + B_i$; $Y_i = (A_i - B_i) \cdot W^k$;
$k$ = Modulo$(k + 2^{step}$, M/2);
Rearrange the index;
NCS_send(THREAD1, my_id, THREAD1, HOST, X, size);
NCS_send(THREAD1, my_id, THREAD1, HOST, Y, size);

}

**Compute_fft2(){**

The algorithm is same as Compute_fft1 except that
$thread\_num = 2 \cdot my\_num + 1$

}

Figure 21: FFT in NCS_MTS/p4 environment

Table 3: Execution times of FFT in seconds

| Nodes | Ethernet | | | NYNET testbed | | |
|---|---|---|---|---|---|---|
| | p4 | NCS_MTS/p4 | % Improvement | p4 | NCS_MTS/p4 | % Improvement |
| 1 | 5.76 | 5.84 | - | 5.25 | 5.32 | - |
| 2 | 5.09 | 4.76 | 6.5% | 3.65 | 3.34 | 8.49% |
| 4 | 4.58 | 4.32 | 5.7% | 2.72 | 2.43 | 10.66% |
| 8 | 3.91 | 3.47 | 11.25% | - | - | - |

Interface (API). The Multithreaded environment allows applications to overlap computation and communication and provide a modular approach to efficiently support applications with a wide range of Quality of Service requirements.

We implemented the multithreaded subsystem and integrated it with p4 parallel/distributed tool. From the benchmark results presented it is clear that multithreaded message passing is a powerful distributed computing paradigm. This programming paradigm allows user to reduce the impact of propagation delay on application performance and support efficiently a wide range of HPDC applications with different QOS requirements.

We are also investigating the performance of NCS_MTS/p4 implementation when p4 is replaced by PVM and MPI. We are currently implementing the second approach for NCS implementation. Once this implementation is complete, we do believe that NCS applications would run at much higher speed than that can be obtained using existing parallel/distributed software tools.

## References

[1] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch, "Low-Latency Communication over ATM Networks using Active Messages", Cornell University, Ithaca, NY 14850.

[2] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", Report # UCB/CSD 92/#675, University of California, Berkeley, CA 94720.

[3] Richard P. Martin, "HPAM: An Active Message layer for Network of HP Workstations", University of California at Berkeley.

[4] David Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", University of Washington, Technical Report UWCSE 93-05-06.

[5] Kleinrock, L. "The latency/bandwidth tradeoff in gigabit networks". IEEE Communication Magazine 30, 4 (Apr. 1992), 36-40.

[6] Hariri Salim, Park Sung-Yong, Reddy Rajashekar, Subramanyan Mahesh, Yadav Rajesh, and Parashar Manish "Software Tool Evaluation Methodology" International Conference on Distributed Computing Systems, 1995.

[7] Edward W. Felten and Dylan McNamee, "Improving the Performance of Message-Passing Applications by Mutithreading", Proceedings of the Scalable High-Performance Computing Conference, April 1992.

[8] Ralph Butler, and Ewing Lusk, "User's Guide to the p4 Programming System", Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801

[9] Shene-Ling Chang, David H.C. Du, Jenwei Hsieh, Mengjou Lin, and Rose P. Tsang, " Parallel Computing Over a Cluster of Workstations Interconnected via a Local ATM Network" University of Minnesota, Minneapolis, MN 55455.

[10] Jongbaek Park and Salim Hariri, "Architectural supoort for high-performance distributed system", Technical Report, Department of Electrical and Computer Engineering, Syracuse University, Syracuse, NY, 1992.

[11] Salim Hariri, "High Performance Distributed Computing: Network , Architecture, and Programming", to be published by Artech House Inc. in 1995.

[12] Adam Beguelin, Jack Dongara, Al Geist, Robert Manchek , and Vaidy Sunderam, "User Guide to PVM", Oak Ridge National Laboratory, Oak Ridge TN 378 31-6367 and Department of Mathematics and Computer Science, Emory University, February 1993.

[13] "Document for a Standard Message-Passing Interface", Message Passing Interface Forum, 1993.