To be presented at the 29th Annual Hawaii International Conference
on System Sciences, January 3-6, 1996, Wailea, Hawaii.

Concurrency, a Case Study in Remote Tasking and Distributed IPC

Dejan S. Milojičić, Alan Langerman, David L. Black, Steven J. Sears,
Michelle Dominijanni, and Randall W. Dean
OSF Research Institute

Abstract

Remote tasking encompasses different functionality, such as remote forking, multiple remote spawning, and task migration. In order to overcome the relatively high costs of these mechanisms, optimizations can be applied at various levels of the underlying operating system or application. Optimizations include concurrent message transmission, increased throughput and reduced latency at the distributed IPC level; batching, overlapping, and pipelining at the remote tasking level; and multithreading at the application level. Of particular interest is the resulting concurrency, since in a complex program, it may be a dominant performance factor.

Distributed IPC is typically characterized by throughput and latency. However, many design and implementation details important for real application performance remain unobserved by this simple characterization. This paper describes distributed IPC from a remote tasking point of view. Remote tasking exercises all aspects of distributed IPC extensively. We analyze two versions of distributed IPC supported in Mach (NORMA IPC and DIPC).

1. Introduction

A traditional monolithic Operating System (OS) may be divided into one or more user-level OS servers and a microkernel. The microkernel provides a small number of fundamental services, such as virtual memory management, interprocess communication, task/thread management, and device drivers. Recently, even more functionality has been moved outside of the microkernel, such as parts of VM handling and device drivers. The servers provide traditional OS functionality, such as the management of processes, files, and networking. Systems like this have been built using Mach [1], Chorus [16], V kernel [6], Spring [10], and others.

The suitability of microkernels for transparent extension to distributed systems has been demonstrated in [4]; IPC, VM, and task management are transparently extended across node boundaries. Distributed IPC provides the ability to send messages to remote nodes; distributed memory management provides mapping and paging from remote nodes; and remote tasking supports remote task creation and migration.

This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC).

In this paper we are concerned with distributed IPC and its effects on remote tasking. The performance of remote tasking is directly dependent on the performance of the underlying distributed IPC, however, our work is concerned not only with improving the latency and throughput, but also with finding and exploiting opportunities for concurrency.

Ideally, concurrency may be introduced at all levels of the system. However, the introduced complexity often does not warrant such a comprehensive approach. We compare how concurrency (or lack thereof) contributes to the performance of remote tasking and distributed IPC for the Mach microkernel. In particular, we explore the impact of concurrent message transmission and reduced port management overhead at the distributed IPC level and various optimizations at the remote tasking level (*e.g.* batching, pipelining, and overlapping) on remote tasking performance.

The goal of distributed IPC in the microkernel is the transparent, seamless extension of local Mach IPC semantics across multiple nodes. NORMA IPC was developed as a proof-of-concept; the DIPC subsystem was developed to improve upon NORMA IPC. NORMA IPC and DIPC are chartered with conveying standard, kernel-mediated Mach messages between nodes, with all the semantic freight that is implied. Mach IPC was not developed to compete with today's low-latency message passing systems (*e.g.*, [8]), which define their own, very light-weight semantics and in many cases avoid interacting with the kernel entirely. The design and implementation of the Mach distributed IPC mechanisms are described elsewhere [9][11].

The goals of remote tasking are to support creation of one or more remote tasks, or migration of the tasks for parallel applications; repartitioning; load distribution; system administration, *etc*.

This paper concentrates on the empirical and quantitative evaluation of the relationship between distributed IPC and remote tasking. We evaluate the trade-offs between efforts and benefits in applying concurrency at different levels.

The remainder of this paper is organized as follows. Section 2 presents an overview of distributed IPC performance improvements and compares it with an earlier design and implementation. These comparisons are continued throughout the paper. In Section 3 we describe

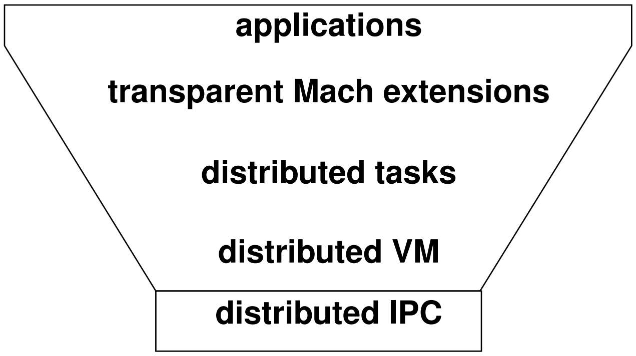


Figure 1. Distributed Services: *higher-level distributed services are built upon distributed IPC. Flow control, throughput, latency, and concurrency are critical issues to solve at this level.*

remote tasking in Mach. Section 4 presents various experiments we conducted. Section 5 compares our work to related research. Conclusions are presented in Section 6.

2. Distributed IPC in Mach

Inter-process communication among cooperating nodes has a long history in the Mach operating system and its predecessors Accent and RiG [15]. Mach IPC was designed to be location transparent; the holders of the send and receive capabilities for a Mach port have no information regarding the location or identity of the task they are communicating with [17]. Thus, Mach IPC messages theoretically may be delivered across node boundaries, transparently. However, the nature of Mach IPC demands kernel interactions to interpret messages, which may carry capabilities and out-of-line regions, transfers of which are mediated by the kernel.

The earliest version of Mach did not contain in-kernel support for inter-node IPC. The extension of Mach IPC over the network was handled by a user-level server called the *netmsgserver* [9]. The *netmsgserver* task resides on all nodes in a cluster and interposes on send and receive capabilities, acting as proxy for receive capabilities on remote nodes. A message passed from sending task to receiving task is transparently intercepted and forwarded by the *netmsgserver*. The multiple *netmsgserver* tasks communicate with one another via standard networking services such as sockets and TCP/IP. Because of the multiple user/kernel boundary crossings necessary in this implementation, the *netmsgserver* implementation of distributed IPC was very slow.

After gaining experience with the *netmsgserver*, an in-kernel distributed IPC service called NORMA IPC was created [2]. NORMA IPC avoids the use of additional servers and kernel interactions, employing simple network protocols to communicate directly between cooperating Mach kernels. The NORMA IPC implementation of distributed IPC became the foundation for higher-level distributed Mach services (see Figure 1). Although ultimately intended for use on the Intel Paragon[5], with a fast interconnect (175 MB/sec) and many

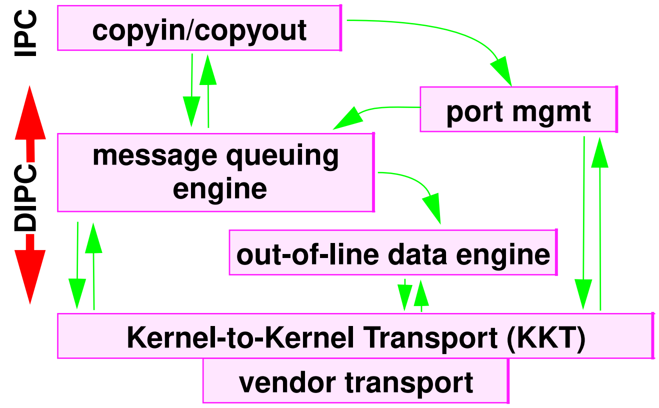


Figure 2. DIPC organization: *the DIPC subsystem consists of several modules, including one for local and remote port management, and for maintaining Mach IPC semantics; one for enqueueing messages on remote nodes and handling flow control issues, while maintaining message ordering; and a third for moving bulk data across nodes. Underneath DIPC, KKT manages reliable, ordered, byte stream connections across nodes.*

nodes, this service was largely developed on Ethernet clusters of PCs and on the Intel Hypercube multicomputer, which had relatively low-speed interconnects (1-3 MB/sec) and relatively few nodes.

The implementation of NORMA IPC implicitly tied Mach IPC semantics to lower-level networking protocol implementations. For instance, NORMA IPC relied on a stop-and-wait network protocol to maintain ordering between and within the bytes of a message. Only one message at a time could be transmitted from a sending to a receiving node. Thus, concurrency for multiple messages was limited to messages sent from a sender to receivers on different nodes. Within a long message, a sending node transmitted a single page at a time to a receiving node, waiting for the receiver to acknowledge receipt of each page. The NORMA IPC design provided for increasing the transmission window size from one page to n pages, but still required a stop and wait synchronization after each block of n pages. This technique limited throughput for single message transmission.

NORMA IPC eventually was ported to the Intel Paragon, with its fast interconnect and scalability to hundreds, even thousands of nodes. In such a high-throughput, low-latency environment, NORMA IPC encountered many flow control, scalability and performance problems due to its design limitations. These limitations led to a complete redesign and reimplementation.

Today's distributed Mach IPC subsystem, DIPC [11], benefited from the experiences gained with the *netmsgserver* and NORMA IPC. DIPC is an in-kernel service, with a well-defined architecture that separates IPC semantics from data transport, permitting a high degree of concurrency as well as fast transmission of individual messages. (Unfortunately, the

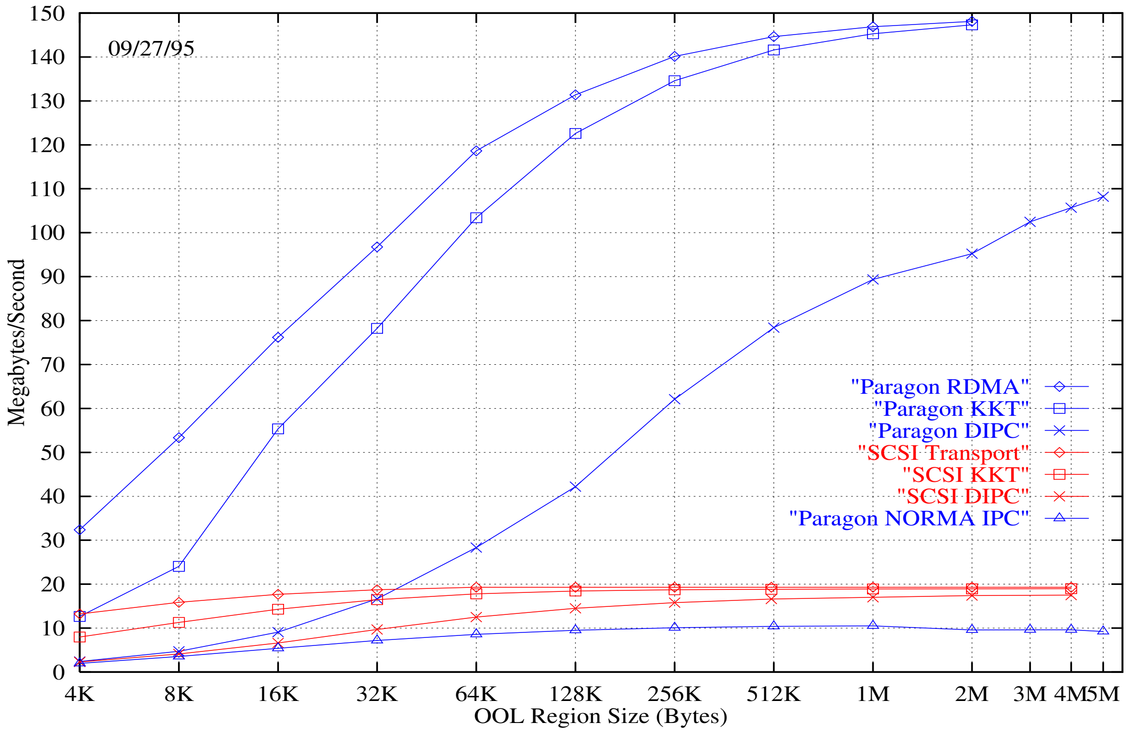


Figure 3. DIPC, KKT and NORMA IPC Throughputs for Paragon and 90Mhz Pentium SCSI Cluster. *Throughput versus message transfer size; each message contains a single out-of-line region of the specified size.*

semantic limitations of Mach IPC prevented the use of message transfer protocols that could operate directly from the sending task to the receiving task without kernel intervention.) The data transport, called Kernel-to-Kernel Transport (KKT), is a re-usable layer available for other clients besides DIPC. See Figure 2. Separation of IPC semantics from data transmission allows easy portability of DIPC across different processors and interconnects, such as the Paragon Mesh (i860), SCSI clusters (Pentium), and Ethernet clusters (i486, Pentium). Flow control mechanisms are designed into DIPC and KKT, preventing receivers from being overwhelmed by senders, even when hundreds or thousands of senders are involved, but allowing considerable concurrency and throughput.

A full description of DIPC and KKT exceeds the scope of this paper, but we can succinctly describe DIPC's concurrency, throughput, and latency properties. Although a specific transport implementation of KKT may place some constraints on message traffic, DIPC allows concurrent transmission of:

- messages in transit from a sending node to different receiving nodes
- messages in transit from a sending node to different ports on a single receiving node
- messages in transit from a sending node to the same port on a receiving node, when multiple threads attempt to receive messages concurrently
- messages in transit between different node pairs

The concurrency provided by DIPC is a significant improvement over that provided by NORMA IPC, dramatically affecting the performance of concurrent remote task creation, as will be seen in Section 4.

We increased throughput for individual message transmission in DIPC by separating Mach IPC semantics from the problem of transporting data. We designed DIPC to avoid imposing constraints on the way in which KKT conveys message data. Thus, high-performance networking protocols may be used, with arbitrarily-large window sizes. DIPC delivers high throughput across its various platforms, noticeably higher than the throughput delivered by NORMA IPC. For example, using 90Mhz Pentium nodes and a fast, wide SCSI interconnect whose sustained peak transfer rate is 19.2 MB/sec, DIPC can saturate the link (see Figure 3). An Intel Paragon, with i860 nodes and a mesh interconnect, achieves a peak transfer rate of 175 MB/sec; the low-level Intel RDMA transport software used by KKT, and therefore by DIPC, has a peak of 147 MB/sec. In this setting, DIPC achieves up to 108 MB/sec, or 73% of available bandwidth. For transfer sizes of 32K and more, KKT provides 75-99% of the bandwidth offered by the lower level protocol, which limits our overall throughput. We have not yet begun to aggressively tune DIPC and KKT; the curves in Figure 3 reveal room for optimization, such as in the overhead of KKT compared to RDMA.

One good question is: why doesn't the shape of the DIPC throughput curve match that of RDMA and KKT, with a fast attack followed by an asymptotic approach to the maximum bandwidth? The answer is that RDMA and KKT are low-level protocols unburdened by significant semantic or operational overhead. DIPC, on the other hand, incurs the overhead of kernel-mediated, local Mach IPC, which leads to large latencies that throttle throughput for small data transfers (see below).

The DIPC design permits latency reduction optimizations that were not possible in NORMA IPC. For instance, an application may optionally disable the No More Senders (NMS) feature in Mach. This feature notifies an application when a receive capability has no more outstanding send capabilities. However, this feature is expensive in a distributed environment: a node with a send capability may have to exchange messages with the node containing the receive capability to update capability counts. Thus, using this feature may increase message latency. As will be seen in Section 4.1, the ability to disable this feature is useful. Other DIPC latency optimizations further reduce the length of the message transmission path.

In Table 1, we compare the time to complete a user-to-user null RPC given two implementations of Mach IPC on the Intel Paragon. (For comparison purposes, we also include performance of DIPC over SCSI). Null RPC latency, in all cases, remains higher than desirable, because of the costs paid for Mach IPC semantics, i860 context switching, user-to-kernel boundary crossings, and as-yet unoptimized DIPC overhead. Worse, as revealed by the local IPC data in Table 1, there is a large difference between the null RPC time when the local IPC hotpath is used and the null RPC time when the hotpath isn't used. In the case of DIPC on Paragon, the null RPC time nearly doubles; and, unfortunately, the design of the hotpath absolutely forbids its use in distributed operation! The DIPC latencies can be further optimized, but will always be handicapped by the local-case penalties outlined above.

Server Location / Mach IPC perform.	NORMA IPC (Paragon)	DIPC (Paragon)	DIPC (SCSI)
Local	220 µsec	318 µsec	115 µsec
Remote	2070 µsec	1272 µsec	1129 µsec

Table 1: Mach IPC over NORMA IPC and DIPC

3. Remote Tasking in Mach

Remote tasking in Mach encompasses remote task creation [3], task migration [13], and concurrent remote task creation (*crtc*) [12]. In each case the majority of the time required to perform the operation is spent in creating remote address space(s) from an address space template on the source node. The address spaces can be:

- created on a single remote node (typically used for UNIX-like remote *fork*),
- created on multiple remote nodes (could be used for the UNIX-like *rfork-multi, rspawn-multi or rexec-multi*), or
- migrated (basis for UNIX process migration).

The algorithm in each case consists of extracting the relevant state describing a task address space, transferring that state from the source node to the destination node, and establishing it there. An address space is described by the state of each VM area (starting address, size, pager capability, *etc*.).

The actual memory pages are not transferred eagerly, but rather on demand. This constitutes the Copy On Reference (COR) optimization, which is a form of lazy evaluation. The advantage of this approach is the low initial cost, which is especially useful for large address spaces. The disadvantage is in increased run-time costs, because delay is introduced by the need to fault in a page from the original node. When creating a remote address space, there are many similarities between remote task creation and task migration.

As part of creating an address space on a remote node, paging paths are established. Remote paging is integrated with distributed memory management and implemented inside the kernel. In order to transfer the task address space state and to establish remote paging paths a number of network messages are required. The optimizations that we performed to the original algorithm for remote task creation consist of eagerly sending more state (batching messages) whenever possible; overlapping messages by eliminating all synchronization points when batching is too complex or requires significant changes to other Mach modules; and relaxing the consistency requirements of VM objects representing read-only mappings, obviating the need for page-out when replicas exist.

The algorithm for concurrent remote task creation is explained in more detail in [12], but for our analysis it suffices to consider it as using a spanning tree to concurrently instantiate address spaces on multiple nodes. First, two tasks are created on two nodes, then these two tasks each create tasks on two subsequent nodes, and so on.

In addition to using spanning trees for creation of remote tasks, we also organize remote paging paths in a hierarchical form. The leaf nodes attempt to page-in from nodes higher in the hierarchy, by traversing the tree until the page is found or root is reached. The hierarchical tree organization trades performance of single node paging for scalability. Due to locality of reference it is probable that some other node on the way to the root of the tree will already have the page.The difference between a flat and a hierarchical paging tree is illustrated in Figure 4.

Hierarchical organization also provides for pipelining since some of the activities can be conducted in parallel on different

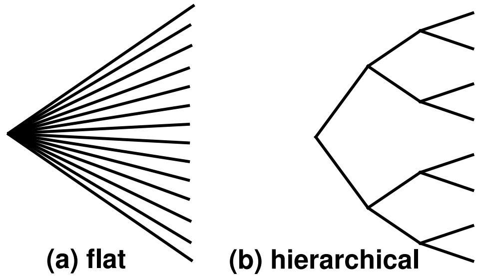


Figure 4. Flat and Hierarchical Paging Trees: *in a flat tree all references point to a single node; in a hierarchical tree references point to the previous node in the hierarchy.*

nodes, further improving performance. Finally, hierarchical paging trees improve scalability because data structures and paging traffic are more evenly distributed among all nodes.

Concurrency in remote task creation can be realized at different levels. For example, when creating multiple remote tasks, concurrency can be achieved at the user level by multithreading the application. Due to communications delay in sending inter-node messages, and the overhead of creating a single remote task, performance can be improved by generating multiple asynchronous requests, each of which creates a single remote task on a different node.

Alternatively, concurrency at the remote tasking level can be improved by optimizing the algorithm for creating a single remote address space, *e.g*., by overlapping and batching of paging path initialization, as described in [12]. Another example of concurrency in remote task creation occurs when task address spaces are distributed using a spanning tree. In this case, concurrency is introduced by pipelined creation on multiple nodes. Such optimizations are complex as they require modifications in the kernel dealing with sensitive kernel structures. Different types of concurrency for remote task creation are summarized in Table 2.

experiment / concurrency type	time overlap	pipelining
flat tree, sequential	NO	NO
flat tree, multithreaded	YES	NO
crtc, hierarchical tree	YES	YES

Table 2: Concurrency in remote task creation

Finally, as described in the previous section, optimizations can be achieved at the distributed IPC level. These optimizations have the most impact because they also benefit other systems relying on distributed IPC. Besides the straightforward optimizations of throughput and latency, concurrency has a significant impact on the performance of higher levels. Experiments show that a lack of concurrency at a lower level can preclude concurrency at higher levels, thereby impairing system performance. The three levels in which concurrency may be introduced for remote tasking are presented in Figure 5.

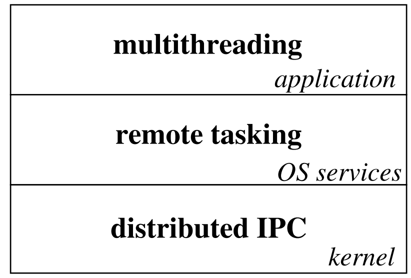


Figure 5. Different levels of concurrency *can be introduced in order to optimize performance. Distributed IPC at the core kernel level; remote tasking at the OS services level; multithreading at the application level.*

4. Conducted Experiments

We compare performance of the creation of multiple remote tasks for sequential, multithreaded, and concurrent remote task creation. Similar conclusions can be drawn for single remote task creation or migration; however, the results are more pronounced for *crtc* because each improvement is multiplied by the number of tasks created, and concurrency may be increased by overlapping and pipelining asynchronous remote task creation operations. We also analyze the contributions of DIPC compared to NORMA IPC.

The measurements reported were collected on a 56-node Intel Paragon multicomputer, running an OSF/1 AD 1.2 operating system server on top of an NMK17.2 microkernel (NORMA IPC) or an NMK19b1 microkernel (DIPC). The tests are Mach level programs, *i.e.*, they do not interact with the OS server, but rather invoke Mach system calls.

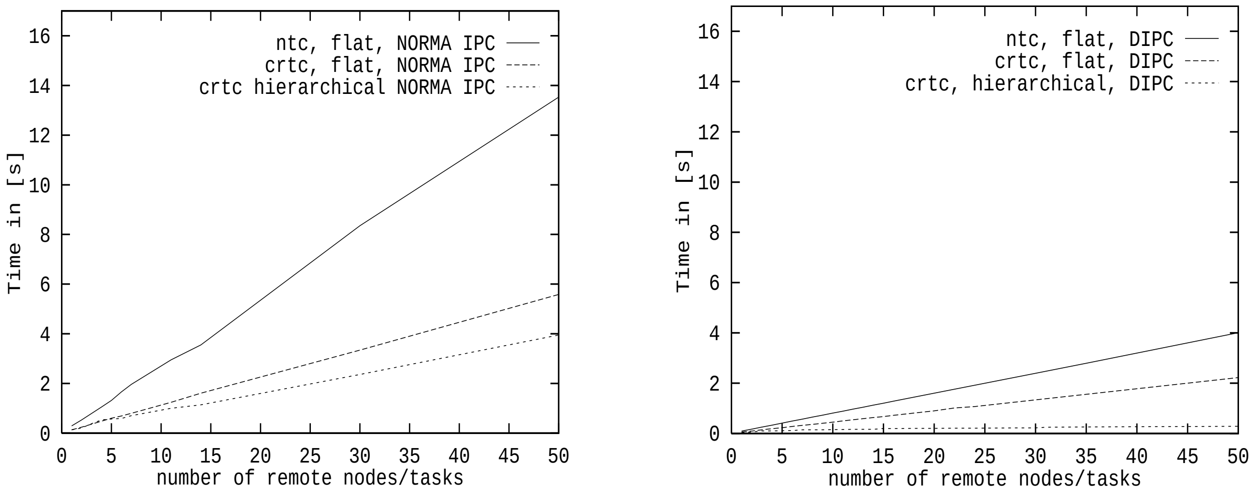
The measurements are grouped in three subsections describing optimizations applied to distributed IPC, remote tasking, and user level programs, followed by a summary. However, many elements are relevant to all of the subsections.

4.1 Distributed IPC Level Optimizations

Figure 6 compares the performance of multiple remote task creation achieved by using:

- a flat task creation tree with the unoptimized single task creation algorithm, *norma_task_create* (*ntc*);
- a flat tree, but using the improved single task creation algorithm embedded in *crtc*; and
- a *crtc* hierarchical tree.

In Figure 6a, we see that the poor concurrency provided by NORMA IPC had a significant impact on the performance improvement of *crtc*. The largest increase in performance is due to the improved algorithm for single remote task creation (2.5 times, *ntc flat vs. crtc flat*), whereas the improvement gained by building a hierarchical tree is less than 30% (*crtc flat vs. crtc hierarchical*). Because of the lack of concurrency



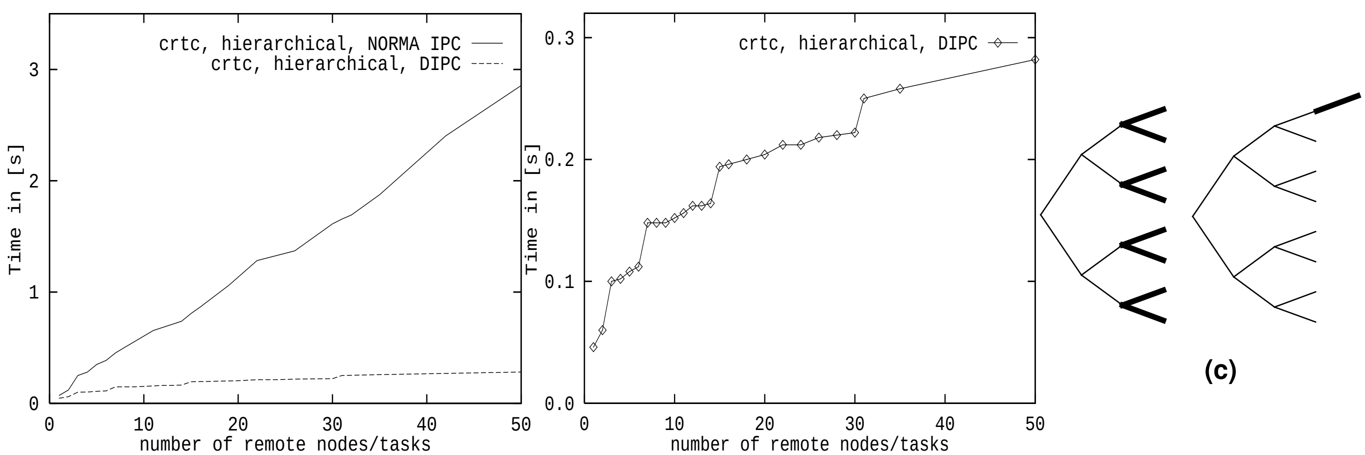
(a) **(b)**

Figure 6. NORMA IPC (a) vs. DIPC (b), for sequential and concurrent remote task creation: *performance of remote task creation for the original single remote task creation algorithm (ntc); the optimized single remote task creation algorithm, with a flat task creation tree (crtc, flat); and concurrent remote task creation with a hierarchical task creation tree (crtc, hierarchical).*

in NORMA IPC, concurrency added in *crtc* cannot be exploited.

In Figure 6b, we see that DIPC not only speeds the creation of a single remote task but also introduces opportunities for additional concurrency. In fact, the concurrency optimizations of *crtc* become more important than its single task creation optimizations. On top of DIPC, the performance improvement for single remote task creation, compared with the unoptimized algorithm, is still significant (~40%) but is relatively less important than in NORMA IPC (*ntc flat vs. crtc flat*). The concurrency optimizations in *crtc* are well matched with DIPC, yielding 8 times better performance than sequential cases of the original algorithm and about 6 times better than sequential invocation of the improved algorithm (*crtc hierarchical vs. ntc flat or crtc flat*).

It is also interesting to inspect the shapes of the *crtc* curves shown in Figure 6 in finer detail, as presented in Figure 7. Figure 7a shows the difference in performance between NORMA IPC and DIPC for hierarchical *crtc*. We magnify the DIPC curve in Figure 7b, where a step function is revealed. Because of the hierarchical nature of remote task creation in *crtc*, we expect to observe similar costs when creating remote tasks at the same depth of the spanning tree. Figure 7c illustrates this proposition: the jump in cost occurs for the 2^n+1 node. NORMA IPC does not show much of a step function because it lacks sufficient concurrency. Attempts to provide concurrency at a higher level fail in this case.



(a) **(b)**

Figure 7. NORMA IPC vs. DIPC, *crtc* Concurrency: *(a) NORMA IPC lacks concurrency so concurrency of crtc cannot be exploited, resulting in a linear curve. (b) DIPC, which allows for multiple window sizes and other concurrent operations, reveals a step function. (c) When adding a node increases the tree depth, performance decreases sharply.*

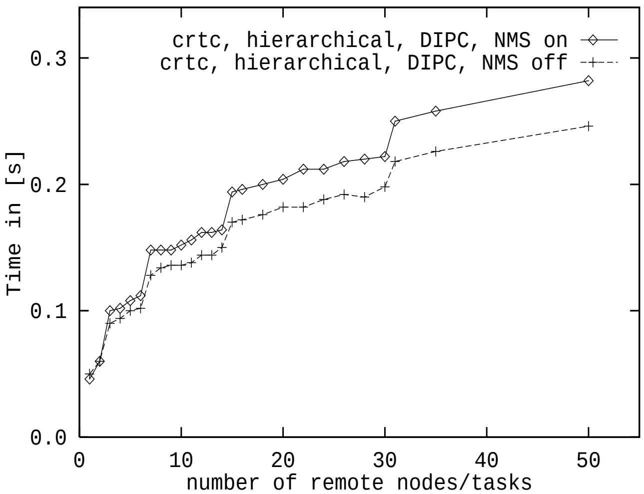


Figure 8. Effect of No More Senders: *an example of a negative influence on the performance of crtc is NMS support, which requires that additional messages be exchanged while crtc is in progress. Disabling NMS yields a latency reduction.*

However, many factors affect performance. One such example is No More Senders (NMS) support (see Section 2). NMS is not needed by *crtc*, so we disabled it. The resulting performance improvement is shown in Figure 8. The use of NMS decreases performance with the logarithm of the number of nodes, because the additional latency accumulates with increasing tree depth. For a larger number of nodes we may experience concurrency problems when many nodes attempt to communicate with the original node, but this was not noticeable in our relatively small Paragon configuration.

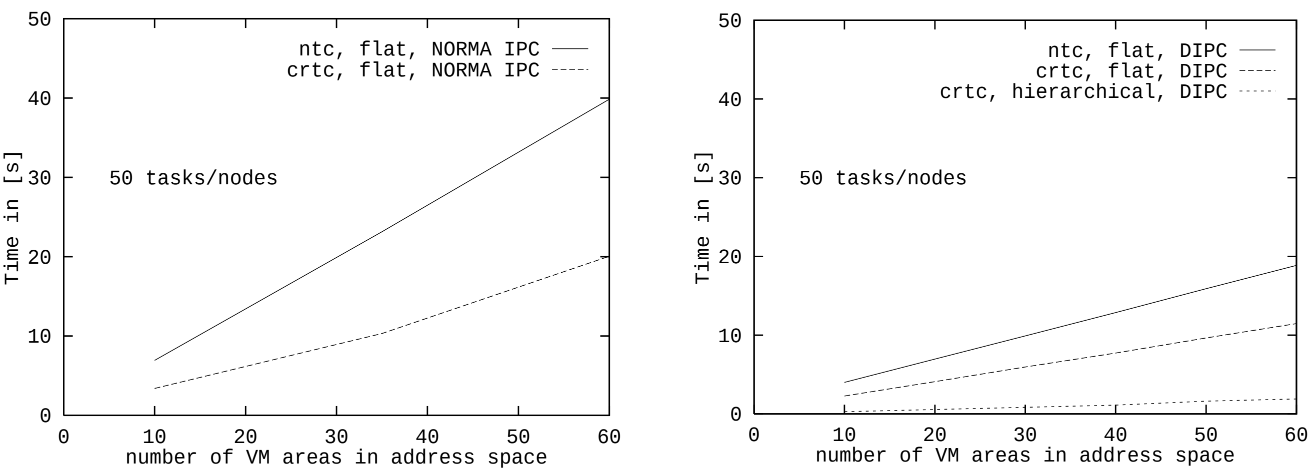
4.2 Remote Tasking Level Optimizations

The improved algorithm for single remote task creation optimizes performance by overlapping multiple pager initialization. The *crtc* further improves performance by simultaneous communication with multiple nodes at the next level in the tree, as well as by overlapping paging initialization between nodes at different depths. Pipelining is achieved by initializing the minimum amount of state at each depth and proceeding with task creation at deeper levels, while performing the balance of the work in parallel.

Similarly to our measurement of the performance of *crtc* as a function of the number of tasks to create, we measured *crtc* performance as a function of the number of VM areas in the source address space. This function is important for sparse address spaces that have many areas. Figure 9 compares the performance of unoptimized, optimized, and hierarchical *crtc* task creation algorithms against NORMA IPC and DIPC. Remote task creation time increases linearly because state must be transferred and a paging path must be initialized for each VM area (each area may be backed by a different pager). The number of VM areas in the task lengthens the phase which has to be finished before remote task creation can proceed to other nodes.

In addition to the creation of remote tasks, we also observed their run-time behavior. We measured the time required to page in 128 pages on one node, as a function of the tree depth (Figure 10) and concurrently on many nodes, as a function of the number of nodes (Figure 11a) in the spanning tree created with *crtc*. Because we create a hierarchical tree, the cost of paging rises as the node is located deeper in the hierarchy, requiring pages to be fetched via intermediate nodes.

The difference between NORMA IPC and DIPC in Figure 10 is entirely due to the difference in single-stream performance of DIPC and NORMA IPC and is not related to concurrency because the experiment is synchronized: pages are faulted in one after another.



(a) **(b)**

Figure 9. *Crtc* vs. number of VM areas: *optimizations at the algorithm level for single remote task creation are dominant over crtc with NORMA IPC. The curves are linear, because each VM area a message to the source node in order to establish the paging path. (The performance of hierarchical crtc in (a) was not measured in a due to a bug in NORMA IPC. The slight change in the slope of the crtc flat curve for NORMA IPC is also an artifact of NORMA IPC.)*

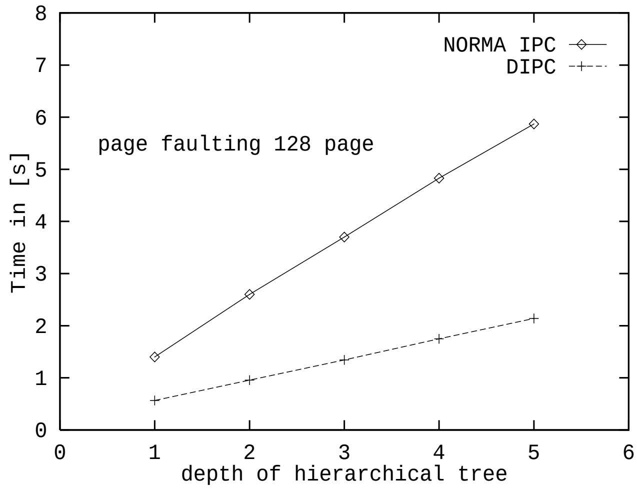


Figure 10. Remote paging vs. tree depth (single node case): *performance of remote paging as a function of paging tree depth is linear for both NORMA IPC and DIPC. Performance penalties are due to the additional hops between the given and the root node.*

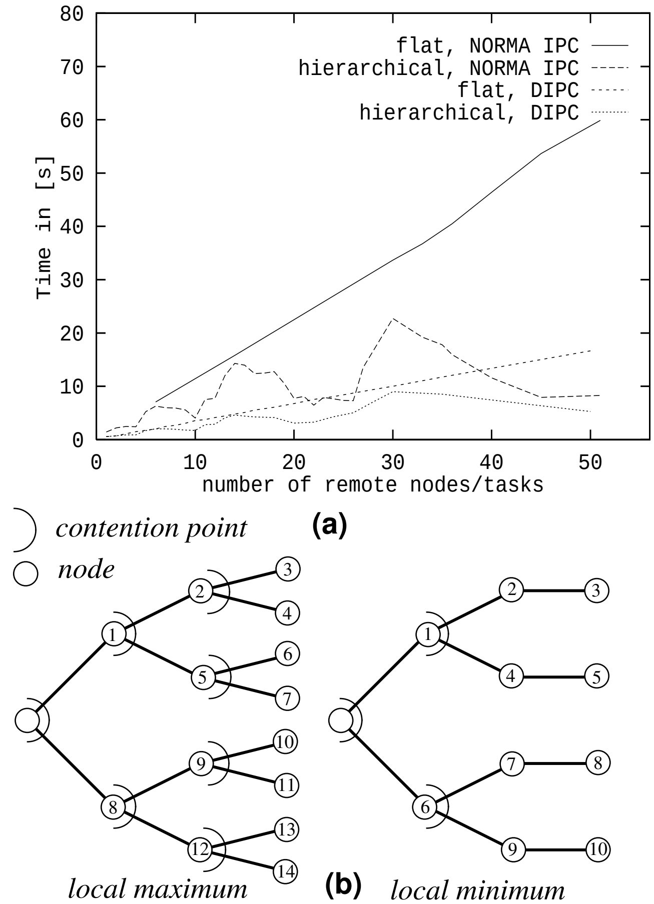


Figure 11. Concurrent Paging from Multiple Nodes: *(a) Page-in of 300 pages from up to 50 tasks (on 50 nodes) for hierarchical and flat tree. (b) Local maxima ($2^{d+1}-2$, d is the tree depth and $d \geq 1$, Figure gives an example for $d = 3$.) correspond to maximum contention; local minima ($2^{d+1}-2-2^{d-1}$)correspond to minimum contention at the leaf nodes.*

Creating a paging hierarchy that parallels the task creation hierarchy reduces contention for paging services, as shown in Figure 11. With a flat paging tree, that is, one node providing paging service for all the others, we have a linear function of page-in time versus the number of nodes. With a hierarchical tree, pages resident on one node in the tree can be used by other nodes in its subtree, thereby eliminating the need to go all the way to the root. In this case we obtain a sinusoid-like function. The curve has alternating local maxima and minima that correspond to the tree configurations as presented in Figure 11b. Maximum performance (the valleys in Figure 11a) is obtained for the trees that exhibit no branching at the first level of nodes above the leaf nodes. Minimum performance (the peaks in Figure 11a) occurs when the tree is fully populated and there is maximum contention at all non-leaf nodes. The example uses a binary tree; in wider trees the contention effects would be even worse, because there will be more leaf nodes and they dominate in costs.

4.3 Remote Tasking vs User Level Optimizations

In the remaining experiments, we are interested in the contribution of user level vs. remote tasking level optimizations, and therefore we shall only measure the performance of remote task creation for DIPC. We analyze remote task creation as a function of the number of nodes/tasks, for sequential creation of a flat tree, multithreaded creation of a flat tree, and hierarchical tree creation.

In Figure 12 we observe that multithreading improves performance over sequential remote task creation by about 20 percent, whereas concurrent remote task creation improves it over 14 times. The step-function performance curve for *crtc,* in hierarchical case*,* demonstrates a significant performance improvement when compared with the performance curves generated by the flat sequential and flat multithreaded algorithms. Another possibility is to create remote tasks using a spanning tree at user level. We did not pursue this goal because the current Mach interfaces do not support it.

Another aspect of concurrency can be observed in the task termination phase. While it is possible to create remote tasks in a concurrent manner, it is not possible to terminate them the same way because the parent/child relationship is not maintained by the kernel. Performance improvement of termination in case of *crtc* is due to the organization of the paging path in the form of a hierarchical tree. In a flat tree, paging path termination on all nodes must interact with a single paging node, whereas in a hierarchical tree, termination is performed between the given node and the node higher in the hierarchy, thereby eliminating the bottleneck with one node. This can be observed in the smoothed out step curve for hierarchical tree termination in Figure 13. Because of the

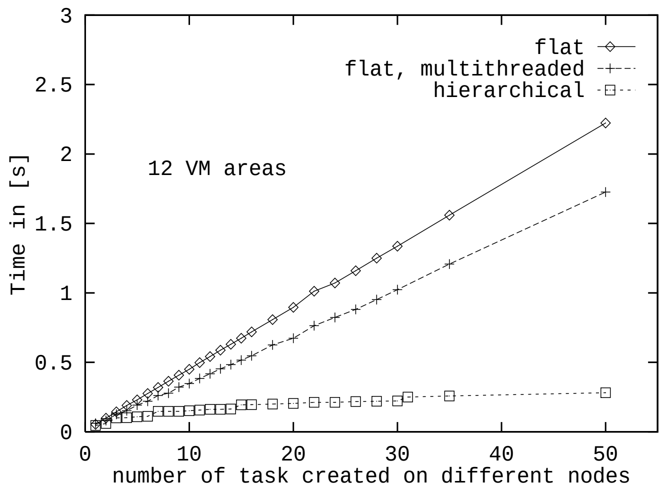


Figure 12. *Crtc* performance as a function of the number of nodes: *sequential creation of a flat tree, multithreaded creation of a flat tree, and hierarchical tree creation.*

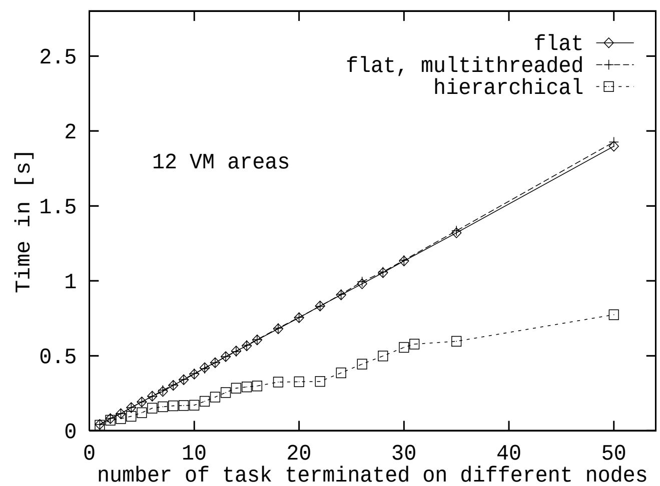


Figure 13. Termination (flat and hierarchical Trees): *due to serialized termination of VM objects, multithreading does not improve performance over sequential termination. Lack of Mach task parent/child relationship, results in little benefit from hierarchical tree.*

serialized nature of the termination phase in the distributed memory management layer, multithreading termination doesn't improve performance as it does in the creation phase. This problem is another example of a lower layer of software, in this case distributed memory management, reducing or eliminating concurrency opportunities at higher levels.

5. Related Work

There is relatively little work reported on the problem of remote task creation from the standpoint of distributed IPC, because the dominating cost consisted of page transfer. We have shown, however, that the costs for remote task creation can become significant in the case of sparse address spaces, multiple task creation, or in the use of *copy on reference*, when there is no initial page copying.

In process migration for the Sprite operating system, the number of remote messages required to achieve process migration was not optimized because migration costs were dominated by page transfer (flushing pages over the network to a server), and because migration was a rare occurrence [7]. Additionally, migration costs are related only to one remote address space creation, whereas for *crtc* the costs are a function of the number of remote tasks.

In the original work on remote tasking for Mach, Barrera attempted to minimize the number of network messages required for process migration [3]. He expected caching to improve performance by reusing existing objects on remote nodes, eliminating the need to optimize performance for the initial creation of a remote address space. Our experience is that initial creation can be very costly in some cases, and therefore it needs to be optimized. For example, when creating multiple tasks, we have observed creation times in the range of minutes for sparse address spaces.

In a version of task migration implemented by Milojicic *et al.*, the authors paid particular attention to optimizing performance by minimizing the number of required network messages [13]. This work was implemented in user space, on top of NORMA IPC.

The NX/2 [14] operating system supported concurrent remote process creation. The address spaces are created by loader in a spanning tree fashion. Performance is function of $\log n$, and it is even further improved by using advanced communication mechanisms. Resulting performance ranged from as high as a fraction of a minute for huge tasks to as low as a couple of seconds. However, beside loading code and data, there are no additional requirements, such as paging path construction. This significantly simplifies the algorithm and allows for performance optimization, but it does not support programs with large address spaces.

In the OSF/1 AD TNC system [18], the authors applied the spanning tree algorithm to the implementation of *rfork-multi*(), a version of the Unix fork() system call that creates child processes on multiple nodes. The advantage of this approach over our work is that the implementation is encapsulated in the servers that implement the distributed Unix on top of the microkernel. Our concurrent remote task creation work generalizes this approach by providing the ability to create multiple remote address spaces as a kernel operation that can be used for operations other than *rfork-multi* (*e.g.*, for systems that want to combine multiple address spaces into a single logical process). Our concurrency work for single remote task creation is complementary, as it optimizes an area not addressed by the *rfork-multi* work.

6. Conclusion and Future Work

In this paper, we have explored the performance of remote tasking and distributed IPC, while paying particular attention to concurrency introduced at various levels of the system. We described a number of experiments that exhibit concurrency in remote tasking and DIPC.

Optimizations at each layer contributed to performance improvement. Concurrency at the lowest level has the largest impact on performance, because it is difficult or impossible to achieve concurrency at higher levels in the face of low-level serialization. While rewriting a fundamental subsystem, such as DIPC, demands a relatively large investment, performance rewards gained at this level are available to all higher levels and so the investment may be amortized over many uses. Higher level optimizations may be easier to achieve and may provide dramatic performance improvement - but only if they can use efficient lower layers of software.

Table 3 compares the reward and effort of introducing concurrency at different levels. Multithreading, requiring a relatively simple effort, provides a benefit of about 20%. Effectiveness of improvements in the remote task creation algorithm depend on the underlying distributed IPC. Improvements for single remote task creation were significant for NORMA IPC, and less significant for DIPC. For concurrent remote task creation, the improved algorithm and hierarchical tree creation (compared to the flat tree original algorithm) were 2 and 3 times better. Finally, improvements at the distributed IPC level contributed the most to performance by allowing *crtc* to exploit all possible concurrency. Performance improvement was over 14 times. While the DIPC project required the largest effort, without this work most of the gains realized by *crtc* would not be possible.

level /characteristics	benefit	effort
user level multithreading	1.2x	1 engineer day
kernel level remote tasking	2-3x	1 engineer year
DIPC	14x	4.5 engineer years

Table 3: Benefit/effort vs. level of concurrency: *more effort for lower level, but greater benefit.*

We have shown that introducing concurrency at the lowest levels of software is vital to permitting concurrency at higher levels. Furthermore, by significantly improving low level performance, previously minor details may play a more important role, as was the case with No More Senders support for concurrent remote task creation. By introducing concurrency at all levels of the system, we significantly improved the overall performance of remote task creation.

Acknowledgments

We would like to thank Joseph Boykin, Fred Douglis, Nikola Serbedzija and anonymous HICSS reviewers for suggestions that have greatly improved this paper.

7. References

[1] Accetta, M., et al., "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference,* Atlanta, GA, 1986, pp 93-112.

[2] Barrera, J., "A Fast Mach Network IPC Implementation", In *Proceedings of the USENIX Mach Symposium*, November 1991, pp 1-11.

[3] Barrera, J., "Odin: A Virtual Memory System for Massively Parallel Processors", *Unpublished Document*, 1993.

[4] Black, D., et al., "Microkernel Operating System Architecture and Mach", *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp 11-30.

[5] Bryant, B., Langerman, A., Sears, S., Black, D., "NORMA IPC: A Task-to-Task Communication System for Multicomputer Systems", *OSF RI Operating Systems, Collected Papers*, vol. 2, October 1993.

[6] Cheriton, D., "Binary Emulation of UNIX Using the V Kernel", S*ummer USENIX Conference,* June 1990, Anaheim, California, pp 73-86.

[7] Douglis, F., Ousterhout, J, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", August 1991, *Software-Practice and Experience*, vol. 21, no 8, pp 757-785.

[8] Greenberg, D., Maccabe, B., McCurley, K., Riesen, R., and Wheat, S., "Communication on the Paragon", Proceedings of the Intel Supercomputer Users' Group. 1993 Annual North America Users' Conference, Oct. 1993, pp 117-124.

[9] Julin, D, "Network Server Design", *Carnegie Mellon University, Mach Networking Group Technical Report,* September 1989.

[10] Khalidi, Y., Nelson, M., "An Implementation of UNIX on an Object-oriented Operating System", *Technical Report, SMLI TR-92-03*, Sun Microsystems, December 1992.

[11] Langerman, A., et al., "NORMA IPC Version Two: Architecture and Design", *OSF RI Operating Systems, Collected Papers*, vol. 3, April 1994.

[12] Milojicic, D., Black, D., Sears, S., "Operating System Support for Concurrent Remote Task Creation", *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 24-29, 1995, pp 273-290.

[13] Milojicic, D., Zint, W., Dangel, A., Giese, P., "Task Migration on the top of the Mach Microkernel", *Proceedings of the third USENIX Mach Symposium*, Santa Fe, New Mexico, April 1993, pp 273-290.

[14] Pierce, P., "The NX/2 Operating System", *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, vol 3, pp 384-390, January 1988.

[15] Rashid, R., "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, November 1986, pp 1128-1137.

[16] Rozier, M., et al., "CHORUS Distributed Operating Systems", *USENIX Computing Systems*, Fall 1988, vol 1, no 4, pp 305-370.

[17] Sansom, R, Julin, D., Rashid, R., "Extending a Capability Based System into a Network Environment", *Proc. of the ACM SIGCOMM '86, Symposium on Communications Architectures and Protocols*, August 1986, pp 265-274.

[18] Zajcew, R., et al., "An OSF/1 UNIX for Massively Parallel Multicomputers", *Proceedings of the Winter USENIX Conference,* January 1993, pp 449-468.