DUNES: A Performance-Oriented System Support Environment for Dependency Maintenance in Workstation Networks*

John Cruz† Kihong Park‡
Network Systems Lab
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{cruz,park}@cs.purdue.edu

Abstract

With the proliferation of workstation clusters connected by high-speed networks, providing efficient system support for concurrent applications engaging in nontrivial interaction has become an important problem. Two principal barriers to harnessing parallelism are: one, efficient mechanisms that achieve transparent dependency maintenance while preserving semantic correctness, and two, scheduling algorithms that match coupled processes to distributed resources while explicitly incorporating their communication costs.

This paper describes a set of performance features, their properties, and implementation in a system support environment called DUNES that achieves transparent dependency maintenance—IPC, file access, memory access, process creation/termination, process relationships—under dynamic load balancing. The two principal performance features are push/pull-based active and passive end-point caching and communication-sensitive load balancing. Collectively, they mitigate the overhead introduced by the transparent dependency maintenance mechanisms. Communication-sensitive load balancing, in addition, affects the scheduling of distributed resources to application processes where both communication and computation costs are explicitly taken into account.

DUNES' architecture endows commodity operating systems with distributed operating system functionality while achieving transparency with respect to their existing application base. DUNES also preserves semantic correctness with respect to single processor semantics. We show performance measurements of a UNIX based implementation on Sparc and x86 architectures over high-speed LAN environments. We show that significant performance gains in terms of system throughput and parallel application speed-up are achievable.

*Supported in part by NSF grant ESS-9806741.
†Additionally supported by a fellowship from the Purdue Research Foundation (PRF).
‡Additionally supported by NSF grants ANI-9714707, ANI-9875789 (CAREER), and grants from PRF and Sprint.

1. Introduction

1.1. Motivation

With the advent of high-speed networks connecting a large number of high-performance workstations via high-speed local area and wide area networks, harnessing their collective power for distributed computing, including parallel computing, has become a viable goal. In addition to intrinsic limitations such as latencies introduced by increased physical distances between networked hosts, two key issues need to be addressed to facilitate a distributed computing environment capable of emulating the prowess of tightly coupled parallel computers—efficient mechanisms for transparent dependency maintenance and resource scheduling which explicitly incorporate communication cost. Although these issues arise in parallel machines as well, their impact is amplified in workstation networks requiring new solutions.

With respect to transparent dependency maintenance, we seek to design efficient mechanisms that allow the flexibility to perform dynamic resource scheduling—in particular, load balancing—when it is deemed beneficial to do so without the mechanism itself becoming a burdensome cost factor. The mechanism should preserve semantic correctness when scheduling tasks across distributed resources and it should be easily deployable, to the extent possible, on commodity computing platforms.

With respect to communication-sensitive resource scheduling, the lack of special calibrated communication facilities in the form of interconnection networks renders a workstation cluster more susceptible to communication overhead including congestion effects. In the case of load balancing, balancing of processor load without proper regard for communication costs can deteriorate performance when network communication becomes a dominant factor.

To harness parallelism in workstation networks effectively, an integrated system support approach is needed that achieves transparent dependency maintenance efficiently and makes load balancing decisions based on explicit consideration of both computation and communication costs.

1.2. New Contributions

The contributions of this paper are twofold. First, we design a system support environment called DUNES (**D**istributed **UN**ix **E**xten**S**ion) which achieves transparent dependency maintenance of coupled processes under dynamic load balancing. DUNES is implemented as a user-level distributed operating system which exports a single system image to the user and achieves single processor UNIX semantics. Second, we design a set of performance enhancement features that mitigate the overhead introduced by the user-level transparent dependency maintenance mechanisms. Communication-sensitive load balancing facilitates an integrated approach to computation and communication control where the scheduling of application processes to distributed resources is affected by explicit incorporation of both computation and communication costs.

1.2.1 Transparent Dependency Maintenance

As part of the distributed OS functionality, our system implements dynamic process migration following the user-level mechanism employed in Condor [15]. However, unlike in Condor, our dynamic process migration mechanism handles *dependencies* arising from interprocess communication, file access, memory access, process creation, and process relationships, maintaining transparent bindings consistent with single processor UNIX semantics[1].

The first form of transparency is *functional transparency* where DUNES ensures that various forms of dependencies including process-to-process and process-to-file dependencies are transparently maintained by the system in the presence of dynamic scheduling. The second form of transparency is *semantic transparency* where, in the process of achieving functional transparency, the semantic correctness of application execution is ensured. DUNES provides a complete *single system image* to the user which extends to semantic correctness. In particular, DUNES preserves single processor UNIX semantics.

1.2.2 Performance Features

Although implementing transparent dependency maintenance at the user-level that achieves single processor UNIX semantics is a nontrivial technical challenge, the resulting support mechanisms would be of limited utility if their associated overhead were significant. We mitigate the overhead cost by employing active and passive end-point caching as part of DUNES' mechanism design. At the algorithmic level, we use communication-sensitive load balancing to affect dynamic scheduling which explicitly accounts for communication costs arising from dependencies.

In the context of concurrent application development for parallel and distributed applications, the *programming model* that DUNES exports to the programmer is one of writing concurrent programs for a single processor UNIX environment.

The rest of the paper is organized as follows. In the next section we summarize related work. This is followed by Section 3 which describes the DUNES architecture including its functional and performance features. Section 4 shows performance results of a DUNES implementation for Solaris UNIX (Solaris 2.6) on both Sparc and x86 architectures measured over high-speed LANs. We conclude with a discussion of our results and future work.

2. Related Work

A myriad of software support environments have been advanced with a view toward facilitating concurrent applications in workstation environments [3, 6, 15]. A principal focus of previous works has been on *enabling technologies* that achieve various forms of transparency including interoperability and ease-of-programming. On the performance side, significant work has been done in load balancing and process migration [1, 9, 12, 18, 20], key components to achieving parallel speed-up and high system throughput.

More recently, performance studies of LAN- and WAN-based systems have shown the importance of

[1] What we mean by *consistent with single processor UNIX semantics* is that, fixing a specific version of UNIX, the execution obtained from our system is sequentially consistent with the corresponding single processor system.

controlling network communication for improving parallel and distributed application performance [2, 5, 13, 14]. The sensitivity of application performance to congestion effects is directly dependent upon the communication/computation ratio and degree of synchrony.

Distributed operating systems, for the most part, are written from scratch and are kernel-based with a well-defined interface to kernel services. The microkernel approach to operating system design tries to make the functionality exported by a kernel minimal with behavioral customizations carried out at the user level. The *library OS* [10] approach to imparting resource management functionalities blurs the dividing line between "hard core" distributed operating systems of the past—by definition, kernel-based—and the abundance of network computing platforms today. In the libOS approach, resource management functionalities are purposely implemented using user-level libraries. Our approach to imparting distributed OS functionality to commodity operating systems can be viewed as an instance of libOS, albeit interfacing with a monolithic kernel rather than a microkernel. We seek the best of both worlds—portability from network computing and efficiency from distributed operating systems.

Our system is similar to Condor [15] in that it follows the latter's user-level process migration scheme. However, unlike Condor, our dynamic process migration mechanism handles *dependencies* arising from interprocess communication, network communication, file access, memory access, and process creation while maintaining transparent bindings consistent with UNIX semantics. Condor is restricted to migrating "standalone" processes with support for remote files access using a network file system.

Another related system is GLUnix [11]. GLUnix, however, does not support process migration and dynamic load balancing. As with Condor, it does not possess the performance enhancement features of DUNES. A similar observation holds for PVM (Parallel Virtual Machine) [19], an execution environment for the development and execution of large concurrent and parallel applications that consist of many interacting, but relatively independent, components. MPVM [4] and DynamicPVM [8] are extensions to PVM that support process migration. They follow the approach used by Condor to checkpoint and restart processes.

3. Structure of DUNES

DUNES (**D**istributed **UN**ix **E**xten**S**ion) is a distributed operating system designed using the approach of *library operating systems* [10]. Distributed OS functionality is injected into a commodity OS—our implementation is in the context of Solaris 2.6—by redefining the service access point or system call interface (mostly wrapper code trapping to `syscall` in Solaris) to kernel services, replacing the system call library with our modified library and relinking applications with the new library. Thus applications need not be recompiled.

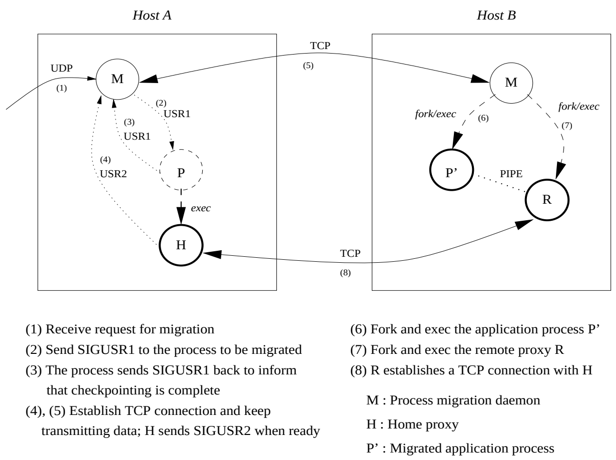

Figure 3.1. Migration of process P from host A to host B and resulting triangle relation between migrated process P' via remote proxy R to home proxy H.

3.1. Functional Features

3.1.1 Transparent Processor Sharing

Process migration As with other distributed operating systems and distributed computing environments, a principal component of DUNES is the transparent enabling of processor sharing across different workstations which facilitates increased system throughput and application performance if the associated overhead is not "too large." The primary enabling feature of transparent processor sharing is *process migration*. Techniques for transparent process migration using user-level libraries for checkpointing and restart are well-known [15, 16, 17] and we follow an analogous strategy in our own implementation.

Basic system configuration A migrating process maintains state information on the host where it was initially started—called the *home base*—in the form of a *proxy process* that subsequently handles its dependencies transparently. This is done by `exec`ing the proxy code from inside the application process—the last action of DUNES' checkpoint routine—which then inherits the relevant properties of the migrated application process. The migrated application process also maintains a corresponding *remote proxy* on the destination host which, in addition to acting as a liaison, also carries out other functionalities including managing the local cache for passive and active end-point caching. The home base proxy, by monitoring all open descriptors belonging to the migrated application process on the home base, is able to transparently handle dependency relations.

If a migrated process is subsequently migrated again, then all state information on the previous remote host is deleted and the new configuration reached is isomorphic to the previous migrated configuration: home proxy on the home base and remote proxy and migrated application process on the destination host. Thus repeated migration does not increase the complexity of the system state. A snapshot of the system—when restricted to the state information for a single migrated process—is shown in Figure 3.1.

3.1.2 Transparent Dependency Maintenance

Functional transparency Transparent process migration, for isolated processes, is a straightforward matter. All processes have some form of dependency (e.g., parent/child relation in UNIX), but more importantly, most processes engage in some form of activity such as file access, interprocess communication (IPC) on a single host, network communication over separate hosts, and shared memory access. Furthermore, a process, after migration, may fork off one or more processes which can complicate the dependency structure significantly.

Semantic transparency Another important aspect of maintaining dependencies transparently is the issue of correctness or semantics. DUNES' functional features provide single processor UNIX semantics. The programmer can write concurrent code with single processor UNIX semantics in mind and the transparent dependency mechanism will guarantee that its execution will be sequentially consistent with the program's execution on a single processor UNIX system.

Transparency mechanism When a process P migrates from a host, it leaves a proxy process H on that machine. On the destination machine a proxy process R is created (cf. Figure 3.1). The application process talks to R through a pipe and R talks to H over the network using TCP. For every system call that a migrated application invokes, a request is sent to R which is forwarded to H. The latter executes the system call on behalf of the application process and sends the result back to the application process through R. As H is `exec`ed by the application process, it inherits file descriptors, signal dispositions, and other relevant properties leaving the dependencies intact for transparent maintenance.

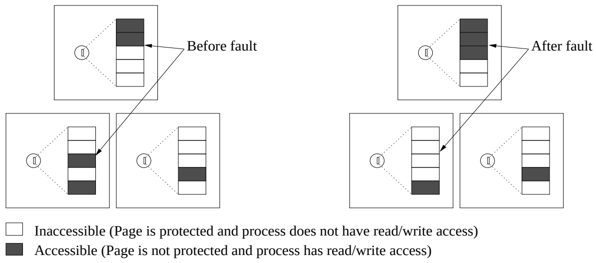

Figure 3.2. Three processes on different machines sharing the same memory segment. Left: Status before an access fault. Right: Status after changing the access permissions and transferring the page

When a migrated process `fork`s, a similar structure (H and R) is created for the child process. This ensures that the child process can be separated from the parent process for further migration. The child is an autonomous entity and the resulting configuration is indistinguishable from the one where the child process `fork`ed before migration.

3.1.3 Shared Memory Segments

DUNES handles accesses to *shared memory segments*—i.e., segments created and shared through `shmget` and `mmap` system calls—transparently after processes have migrated. Memory segments created through `mmap` which are marked *private* are no different from data segments. Due to this, data consistency and integrity follow immediately.

Figure 3.2 illustrates the basic mechanism. A process has access to only those pages in its virtual address space that are marked accessible. Any access to a page that is marked inaccessible will result in a segmentation fault, which is caught by a signal handler. If the access is to a valid address, DUNES identifies the process that has possession of the page and a message is sent to the page owner to relinquish the page: in essence, set the page as inaccessible. It is then transferred to the process that faulted. When control returns from the signal handler, the process continues with access to the page restarting at the instruction that caused the segmentation fault.

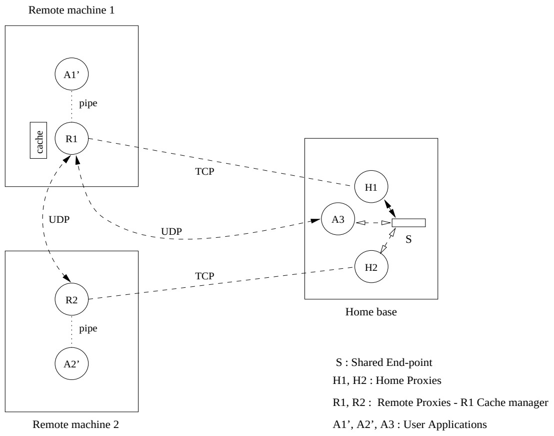

Figure 3.3. Active end-point caching.

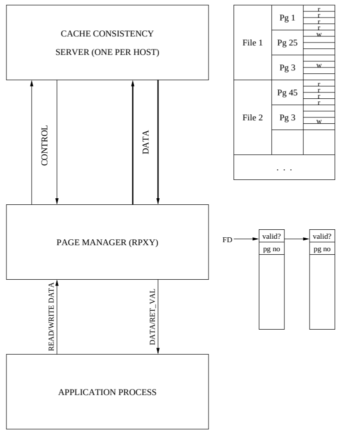

Figure 3.4. Passive end-point caching. The proxy process on the destination machine RPXY acts as the page manager.

3.2. Performance Features

3.2.1 Active End-Point Caching

When communicating processes on a single host are split apart onto separate hosts, or processes engaging in network communication are migrated to more "distant" hosts—either physically (e.g., link latency and physical bandwidth) or logically (e.g., queuing effects and available bandwidth)—then even though the resulting action may yield a net gain in system throughput and application completion time, the performance benefit may be further improved if the effective communication cost is reduced by employing a form of push-based caching. For example, in the case of `fifo` or `pipe` based IPC manifesting as network communication due to process migration, whenever a `write` is executed, the data is immediately shipped to the reader—in the case of multiple readers to the most "likely" reader—such that when a reader executes a `read` operation, the data is already in the reader's local cache and access time is close to the cost of a *local* `read`.

Figure 3.3 shows a typical scenario involving a read-shared `fifo` accessed by two or more migrated processes. One of the migrated processes' proxy takes charge as the cache manager. The other proxies contact the cache manager to get the cached data. When applications running on the home base want to access data from the active end-point, they too have to contact the cache manager. This access subsequently disables caching to reduce the overhead for processes running on the home base.

3.2.2 Passive End-Point Caching

Analogous to active end-point caching, passive end-point caching uses a pull-based or prefetching mechanism to hide communication latency when files are accessed remotely by a process due to migration. Network file systems (e.g., Sun NFS) employ caching to reduce access times. DUNES, by default, does *not* assume the existence of a network file system for generality—optimizations that interface with particular network file systems, if present, are in development—but rather engages in its own push-based caching scheme. We use a system with k pages, page size S, and LRU page replacement policy. The overall structure is shown in Figure 3.4.

DUNES implements single writer/multiple reader semantics using a cache consistency manager which achieves single processor UNIX semantics. The granularity of access is on a per-page basis, and as is generally the case, there is a trade-off between page size and frequency of conflict, with increased granularity carrying a commensurate management overhead cost. Single writer/multiple reader semantics is the simplest but also most restrictive cache consistency protocol.

3.2.3 Communication-Sensitive Load Balancing

A principal lesson learned from dynamic load balancing is that processes best suited for migration are those that are independent, long-lived, and small in size. When this is not the case, the gain obtained from a more balanced load can be outweighed by the resulting amplification of communication cost as well as the overhead associated with process migration itself.

Critical to the success of communication-sensitive load balancing is a method for cost/benefit analysis that accurately estimates or predicts the "goodness" of the configuration reached after the execution of an action that may involve one or more process migrations. This, in turn, is dependent upon an effective measure of goodness. We define such a measure called *progress rate* which incorporates both communication and computation requirements—as exhibited by a process' behavior—which is related to the communication/computation ratio of a process. With the assistance of our run-time monitoring mechanism, we are able to predict the progress rate of a potential next configuration, and by comparing with the *measured* (or *observed*) progress rate of the current configuration, determine the ranking of candidate actions and decide whether it is worthwhile to take an action.

The communication-sensitive load balancer—centralized or distributed—uses the predicted progress rate of candidate configurations and iteratively takes actions until no further performance improvement is deemed possible. The progress rate estimation procedure is accurate as long as actions involving process migrations are *non-overlapping* and admits an efficient form of distributed control. A formal description of progress rate based dynamic load balancing and its properties can be found in [7]. Due to space constraints, we describe its relevant features in Section 4 in conjunction with performance results.

4. Performance Measurements

4.1. Experimental Set-Up

The experiments described in the following sections were conducted on dedicated LAN clusters in the Network Systems Lab (NSL) which is equipped with ten x86-based machines, each with a Pentium II processor at 399 MHz running SunOS 5.6, and four UltraSparc 1+ workstations running SunOS 5.5.1. These machines are connected via two 100 Mbps FastEthernet switches—one connecting the ten x86 machines and the other connecting the Sparc workstations. Some experiments requiring more machines were conducted in a separate lab equipped with twenty x86 machines, each with a Pentium processor at 90 MHz, running SunOS 5.5.1. These machines are connected via a 10 Mbps Ethernet.

All times reported in this section are wall clock times measured using the `gettimeofday` system call with microsecond granularity. For some test cases such as when measuring the performance of `read` and `write` system calls where the cost depends on transient effects—e.g., availability of data in the local cache managed by DUNES—the performance cost measurements were amortized by repeating the operation a number of times (by default 100).

Method	open (ms)	lseek (μs)	read (μs)	write (μs)	fork (ms)	
					parent	child
raw	0.09	27.7	22.6	24.2	1.36	5.41
disabled	0.13	47.7	42.0	46.0	5.59	9.06
enabled	0.96	641.5	618.3	622.9	5.64	9.14

Table 1. System call execution time for processes on home base (i.e., no migration) in single host configuration.

4.2. Overhead of DUNES' Functional Mechanism

4.2.1 Overhead Associated with Encapsulation of System Calls

We identify the pure overhead incurred by DUNES as a software layer. As system calls in DUNES are encapsulated by our modified library, there are overhead costs involved when system calls are invoked. This is shown in Table 1. The caching mentioned in the table corresponds to passive end-point caching. When caching is enabled, processes initiated on the home base have to contact the cache consistency manager when performing file access, thereby increasing the access times. In all other cases, the overhead observed are due to the interposed wrapper code to `syscall`. We note that this interposed code, which allows system calls to implement DUNES' functional and performance features, almost doubles the cost of system calls.

Method	open (ms)	lseek (μs)	read (μs)	write (μs)	fork (ms)	
					parent	child
raw	0.09	27.7	22.6	24.2	1.36	5.41
disabled	1.24	172.3	674.8	652.4	84.69	32.64
enabled	2.05	5145.3	227.1	215.8	71.47	29.92

Table 2. System call execution time for migrated processes in two host configuration.

4.2.2 Cost of System Calls

Passive end-point caching Table 2 compares the cost of executing system calls with and without passive end-point caching. When caching is *disabled*, each system call invocation incurs an overhead of sending messages to the home proxy to fetch data. The second row of Table 2 reflects this overhead. When caching is

Caching of active end-points disabled		
Method	read	write
lcl write/lcl read	22.0	22.8
rmt write/lcl read	1070.7	1073.8
lcl write/rmt read	992.8	23.1
rmt write/rmt read	1530.3	1531.1

Table 3. `read` and `write` system call execution times (in microseconds) for active end-points with caching disabled. *lcl* refers to a process running on the home base (without migration) and *rmt* refers to a process running on a remote machine after migration.

enabled, the third and fourth columns of the third row of Table 2 show a cost reduction of `read` and `write` operations by a factor of 3. If the underlying network is slow or congested, the benefit of caching is further amplified. For system calls such as `open`, the home proxy needs to be contacted to keep the system in a consistent state. This increase in cost can be seen in the same table. When file offsets are shared by processes, each file access incurs an additional overhead of updating the offset maintained at the cache consistency manager. This effect can be discerned for the `lseek` system call. The cost of `fork` for a migrated process is about 6 times as high as a non-migrated process as it involves three separate `fork`s and their initialization.

Active end-point caching Tables 3 and 4 summarize the performance results for active end-point caching. Active end-points are cached using a push-based scheme. When data is available, it is pushed over to the host where the application process resides and then cached locally. Only `read` system calls benefit from active caching as writes have to be flushed immediately.

Without the presence DUNES, a `read` system call takes 8.4 μs and a `write` system call takes 7.6 μs to execute for a payload size of 32 bytes. The row with *lcl write/rmt read* in Table 4 shows the effect of caching. Without caching, the cost of a `read` system call is 992.8 μs and with caching, it reduces to 173.8 μs which is a speed-up of 6. All other values are the same for both the enabled and disabled cases.

Caching of active end-points enabled		
Method	read	write
lcl write/lcl read	24.8	21.6
rmt write/lcl read	1069.0	1073.8
lcl write/rmt read	173.8	23.7
rmt write/rmt read	1508.7	1509.2

Table 4. `read` and `write` system call execution times (in microseconds) for active end-points with caching enabled. *lcl* refers to a process running on the home base (without migration) and *rmt* refers to a process running on a remote machine after migration.

4.3. Communication-Sensitive Load Balancing

4.3.1 Set-Up

Figure 4.1 shows the set-up used in the next experiments. There are three collaborating processes (G_w, G_{rs} and G_{rl}) and one independent process (I) that are initially together on the same host. G_w is a producer that performs some computation and writes the result to G_{rs} and G_{rl}, the consumers. The latter read the data written by G_w and perform some computation. The data rate among these processes is not uniform. The data rate between G_w and G_{rs} is 10 times that of G_w and G_{rl}. Process I does not communicate with either of the three processes. It has a CPU utilization of 0.65 when run in isolation.

4.3.2 Heuristic Load Balancing

We implement three heuristic load balancing schemes that are successively more sensitive to communication costs. All three algorithms have the same structure as shown in Figure 4.2. The algorithms collect load information from the participating machines and identify the least utilized and most utilized machines, based on CPU utilization, to be the target and source machines, respectively, for migration. If the difference in CPU utilization on these two machines differs by an amount greater than a threshold θ_1, then we choose a process from the source machine for migration. Each heuristic has its own way of selecting a process for

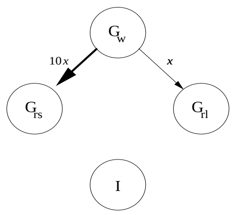

Figure 4.1. Process communication structure. G_w is a producer which writes to G_{rl} and G_{rs}. The amount of writes to G_{rs} is 10 times that of G_{rl}. I is a process running in isolation.

```
GetLoad()
Src = getHostWithHighestLoad();
Dst = getHostWithLeastLoad();
If (highest_load - least_load < THETA1) {
      return;
}
victim = chooseProcessToMigrate();
If (estimated_new_diff(Src, Dst) > THETA2) {
      issueMigrateRequest(victim);
}
```

Figure 4.2. The template code used by the heuristic load balancing algorithms.

migration—`chooseProcessToMigrate()`—and that is where they differ. After a process is chosen for migration, all schemes perform a final check before initiating migration. This is accomplished by testing if the predicted difference in CPU utilization (i.e., *imbalance*)—obtained by subtracting the CPU utilization of the migrating process from the source machine and adding it to the target machine, then taking their resulting difference—is greater than the current imbalance by some threshold θ_2. If not, the migration is cancelled. The rationale behind this test is based on the progress rate based load balancing algorithm where a more comprehensive cost/benefit analysis is performed prior to process migration. Following are summary descriptions of `chooseProcessToMigrate()` for the three algorithms.

Algorithm 1 This is the simplest scheme which does *not* incorporate communication cost information. I.e., it makes load balancing decisions based on CPU utilization only. In particular, it chooses a process which has maximum CPU utilization.

Algorithm 2 This algorithm makes use of both CPU utilization and communication cost where the latter is captured by the *coupling count* of a process—the number of file descriptors open in the process. *Algorithm 2* tries to select a process which is as loosely coupled or independent as possible. The key assumption is that, the greater the number of open descriptors, the tighter its coupling. The specific criterion is based on *average normalized rank* where, given a set of nonnegative numbers $a_1, a_2, \ldots, a_k$, the *normalized rank* γ_{a_i} of a_i, $i \in [1, k]$, is defined as

$$\gamma_{a_i} = \frac{a_i - a_*}{a^* - a_*}$$

with $a_* = \min_{j\in[1,k]} a_j$ and $a^* = \max_{j\in[1,k]} a_j$. Given two sets of numbers and a pair of numbers a_i, b_j belonging to their respective sets, the average normalized rank $\bar{\gamma}_{a_i,b_i}$ is defined as

$$\bar{\gamma}_{a_i,b_i} = \omega\gamma_{a_i} + (1-\omega)\gamma_{b_i}$$

where $0 \leq \omega \leq 1$. We form the two sets by choosing the CPU utilization numbers and coupling count (reordered) of processes belonging to the source host with $\omega = 1/2$. We then compute the average normalized utilization for each process and choose the process with a maximum value.

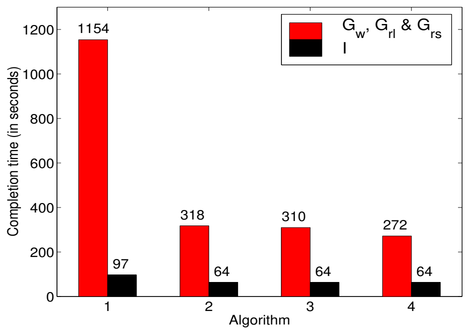

Figure 4.3. Completion times with various algorithms on a three host scenario. The process structure is given in Figure 4.1.

Algorithm 3 We observe that *Algorithm 2* assumes that the number of open file descriptors is proportional to the amount of communication. Clearly, this need not be true. A process could have many file descriptors open and may not use any of them. Similarly, a process could have just one connection open through which it executes frequent data transfer. To capture the quantitative data rate, *Algorithm 3* uses the runtime monitored per-descriptor data rate to compute the total data rate of a processes. It then chooses the process with a maximal average normalized rank where the latter is computed using the total data rate in place of coupling count.

4.3.3 Performance of Algorithms 1, 2, and 3

We compare the three schemes in the same setup described above with the four processes G_w, G_{rl}, G_{rs} and I, all initiated on a single host. We examine the actions of the three algorithms for a three host scenario where on one host the four processes are initiated and the other two are idle.

We observe that *Algorithm 1*—which only considers CPU utilization—selects G_w to migrate. G_w happens to be the most strongly coupled process, but also the one with the highest CPU utilization, which renders it least suited for migration due to the dominance of

```
main() {
    readInput();
    for(;;) {
        updateX();
        sendUpdates();
        receiveUpdates();
        barrier_sync();
        diff = computeDiff();
        if (terminate(diff))
            break;
    }
}
```

Figure 4.4. Sample code used in the parallel iterative algorithm to solve system of linear equations.

communication cost. After migration, the CPU utilization of G_{rl}, G_{rs} decrease significantly due to their dependence on G_w, prompting *Algorithm 1* to migrate I next. *Algorithm 2* selects I and G_{rs} to migrate to the idle hosts, in that order, and after I terminates, it migrates G_{rl}. *Algorithm 3* initially moves I and G_{rl} to the two idle hosts, and upon completion of I, migrates G_{rs}. Although *Algorithm 3* yields the best performance, the optimal decision—shown as *Algorithm 4* in Figure 4.3—is not to migrate G_{rs} after I terminates. This shows the potential for further performance improvement by incorporation of refined communication and computation information.

4.4. Parallel Iterative Linear Equation Solver

4.4.1 Problem Domain

In this section, we show the performance of DUNES at facilitating parallel distributed computing for solving linear equations. Consider the problem of solving a system of linear equations $Ax + b = 0$ where $A = (a_{ij})$ is an $m \times m$ matrix. Finding a numerical solution for x can be formulated as a fixed point problem which can be solved by the iterative procedure

$$x_i := -\frac{1}{a_{ii}} \left(b_i + \sum_{j=1}^{i-1} a_{ij} x_j + \sum_{j=i+1}^{m} a_{ij} x_j \right).$$

If the spectral radius of A is less than 1, the iteration can be shown to converge.

A generic sample code used for implementing the iterative procedure is shown in Figure 4.4. After initialization, there is a loop within which a computation phase is followed by a communication phase. At the end of the latter, a `barrier` call for synchronization followed by a termination check are executed. Given n processes, each process is assigned m/n variables which it is responsible for updating. The updated values are then mutually exchanged using regular IPC (e.g., `fifo`). The `barrier` function call contacts a barrier server process and returns when the server process sends back a "go-ahead" message after synchronization.

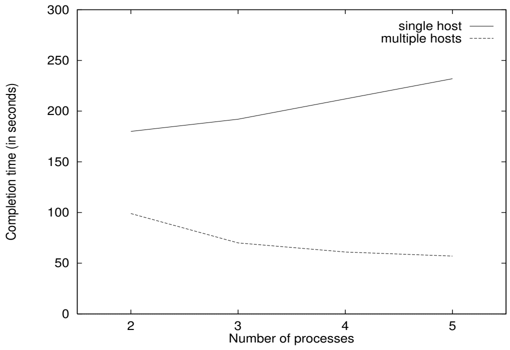

Figure 4.5. Application completion time as a function of the number of processes participating in the computation when all processes are scheduled on a single host vs. when each process is scheduled on a separate host.

4.4.2 Parallel Application Speed-Up

Figure 4.5 shows application performance measured by average completion time as a function of granularity—i.e., number of processes—when all processes are scheduled on a single host vs. when each process is scheduled on a separate host by DUNES via process migration. The matrix in the benchmark problem instance tested was of size (3000×3000). The top plot for the single host schedule shows an increase in completion time as the number of processes is increased which is due to the overhead caused by IPC between processes on a single host. The bottom plot shows the completion times when DUNES is allowed to schedule processes by migrating each process to a separate host.

5. Conclusion

We have described DUNES, a library distributed operating system, organized around its two principal features—transparent dependency maintenance and performance enhancement features aimed at (1) mitigating the former's overhead and (2) exploiting its facilitation of dynamic load balancing of coupled applications. We have shown that user-level system support for dynamical scheduling of coupled processes is feasible with active/passive end-point caching reducing the communication cost associated with dependency maintenance, and communication-sensitive load balancing affecting improved performance with respect to application completion time and system throughput. The main thrust of future work is directed at extending the communication-sensitive load balancing model to incorporate real-time CPU scheduling—built on top of Solaris RT mode—to facilitate both guaranteed and best-effort services to time-constrained and QoS-sensitive applications.

References

[1] M. Ashraf Iqbal, J. H. Saltz, and S. H. Bokhari. A comparitive analysis of static and dynamic load balancing strategies. In *Proc. Int. Conf. on Parallel Processing*, pages 1040–1047, 1986.

[2] C. Cap and V. Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, Apr. 1989.

[4] J. Casas, D. L. Clark, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing systems: the journal of the USENIX Association*, 8(2):171–216, Spring 1995.

[5] A. Cheung and A. Reeves. High performance computing on a cluster of workstations. In *Proc. First International Symp. on High-Performance Distributed Computing*, pages 152–160, 1992.

[6] H. Clark and B. McMillin. Dawgs—a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, 1992.

[7] J. Cruz and K. Park. Towards performance-driven system support for distributed computing in clustered environments. Technical Report CSD-TR-98-035, Department of Computer Sciences, Purdue University, 1998.

[8] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic load balancing on parallel systems. In W. Gentzsch and U. Harms, editors, *High Performance Computing and Networking*, pages 273–277, Munich, Germany, Apr. 1994. Springer Verlag, LNCS 797.

[9] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, Nov. 1989.

[10] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 251–266, 1995.

[11] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a globale layer Unix for a network of workstations. To appear in *Software: Practice and Experience*, 1998.

[12] A. Hac and T. Johnson. Sensitivity study of the load balancing algorithm in a distributed system. *Journal of Parallel and Distributed Computing*, 10:85–89, 1990.

[13] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. In *Proc. 3rd IEEE International Symposium on High-Performance Distributed Computing*, pages 211–218, 1994.

[14] A. Heddaya and K. Park. Congestion control for asynchronous parallel computing on workstation networks. *Parallel Computing*, 23:1855–1875, 1997.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8'th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[16] K. I. Mandelberg and V. S. Sunderam. Process migration in UNIX networks. In *USENIX Technical Conference Proceedings*, pages 357–363, Dallas, TX, Feb. 1988.

[17] S. Petri and H. Langendörfer. Load balancing and fault tolerance in workstation clusters – migrating groups of communicating processes. *Operating Systems Review*, 29(4):25–36, Oct. 1995.

[18] J. Smith. A survey of process migration mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

[19] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, practice and experience*, 2(4):315–339, Dec. 1990.

[20] S. Zhou and D. Ferrari. An experimental study of load balancing performance. In *Proc. IEEE Int. Conf. on Distr. Processing*, volume 7, pages 490–497, Sept. 1987.