Process Tracking for Parallel Job Control

Hubertus Franke, José E. Moreira, and Pratap Pattnaik

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598-0218, USA
{frankeh,jmoreira,pratap}@us.ibm.com
http://www.research.ibm.com

Abstract. Job management subsystems in parallel environments have to address two important issues: (i) how to associate processes present in the system to the tasks of parallel jobs, and (ii) how to control execution of these tasks. The standard UNIX mechanism for job control, process groups, is not appropriate for this purpose as processes can escape their original groups and start new ones. We introduce the concept of *genealogy*, in which a process is identified by the genetic footprint it inherits from its parent. With this concept, tasks are defined by sets of processes with a common ancestor. Process tracking is the mechanism by which we implement the genealogy concept in the IBM AIX operating system. No changes to the kernel are necessary and individual process control is achieved through standard UNIX signaling methods. Performance evaluation, on both uniprocessor and multiprocessor systems, demonstrate the efficacy of job control through process tracking. Process tracking has been incorporated in a research prototype gang-scheduling system for the IBM RS/6000 SP.

1 Introduction

The job management subsystem of a computing environment is responsible for all aspects of controlling job execution. This includes starting and terminating jobs as well as the details related to their scheduling. Parallel jobs executing on a distributed or clustered system are typically comprised by a set of concurrently executing tasks that collaborate with each other. Traditional parallel jobs in the scientific computing community (*e.g.*, MPI and HPF programs) consist of a fixed set of tasks, each comprised of a single process. However, we are observing that in the RS/6000 SP users are beginning to write their parallel programs with each task consisting of a set of processes, exemplified by the following situation: A *csh* script fires two collaborating *perl* scripts, connected by a pipe. Each of these *perl* scripts then executes a few different *Fortran* or *C++* programs to perform a computation. In this situation, a task is no longer a single process but a dynamically changing tree of processes. In fact, in the more general case a task can be a forest of processes. Figure 1 shows a parallel job consisting of three tasks. Each task in turn consists of a set of processes.

Many different approaches to controlling the execution of tasks are described in the literature [4]. In such systems, schedulers are typically organized in a two-tier structure, with a central global scheduler for the system and a node-level scheduler (NLS) in each node. The central scheduler is responsible for resource allocation and job distribution.

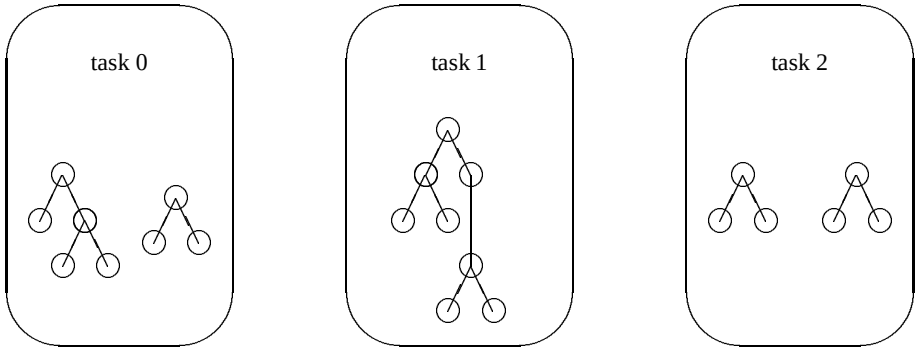


Fig. 1. A 3-way parallel job, each task a forest of processes.

The node-level scheduler is responsible for actually controlling the execution of the individual tasks of that job on its corresponding node. This paper focuses on the issues related to local task control, as performed by the NLS.

Depending on the level of inter-NLS synchronization, different scheduling variations are possible. Explicit gang-scheduling systems always run all the tasks of a job simultaneously [5–10, 17, 21]. In contrast, communication-driven systems [3, 19, 20] are more loosely coupled, and schedule tasks based on message arrival. Independent of the inter-task scheduling approach, we address those cases in which all processes *within a task* are closely coupled, like the *csh* example of the first paragraph. In these situations, all processes of a task must be enabled for execution to allow effective progress within the task.

Typical UNIX operating systems (OS) have their own process scheduling semantics, that aim at improving interactive response times and are based on process priorities. The OS schedulers operate at very fine granularity (order of 10ms) and are very limited with respect to outside control. These schedulers are not geared to the task-centric scheduling required in parallel job control. The traditional approach to bypass the lack of external scheduler support on OS schedulers is to dedicate an entire node (OS image) to the execution of a single task within a parallel job [11, 18, 13]. A more flexible approach would permit multiple tasks to share a node, both in space and time.

Our goal is to develop a task control system that can give each task the illusion that it is running on its own virtual machine. The virtual machines are created by slicing a physical machine along the time axis. Within each virtual machine, all processes of a task are controlled and executed by the OS scheduler. The time-slices for these virtual machines have to be large enough to amortize the cost of context switching. Production parallel applications are typically memory hungry, and it can take quite a while for them to bring their working data from paging space. In fact, simple time-sharing (through the OS scheduler) of parallel jobs can easily lead to trashing of the system. While it is active, each task must also have, for performance and security reasons, dedicated access to the parallel machine's communication device. Context-switching of these devices is also an expensive operation, as reported in [7]. For the above reasons, the granularity of

the task scheduler is therefore much larger (order of seconds or even minutes) than that of the underlying OS. This large granularity can be tolerated in our environment as we target noninteractive applications.

To effectively implement this virtual machine concept, the node-level scheduler must be able to control tasks as single entities. All the processes in a task must be suspended or resumed on occasion of a context switch. There cannot be "stray" processes that either are not suspended when the task is suspended or not resumed when the task is resumed. This behavior must be enforced so that we can guarantee there is no interference between suspended and running tasks. In addition, the time interval it takes to suspend or resume all the processes in a task must be small compared to the time-slice for the task. Finally, we must have a mechanism that precisely defines the set of processes that belong to a particular task of a parallel job. To that purpose, we introduce the concept of *genealogy* of processes.

Each task of a parallel job starts as a single process, which defines the root of that task. This root process can create new processes, which are its direct descendants and belong to the same task. The process creation can continue recursively and all descendants of the original root of the task belong to the same task. Belonging to a task is a genetic property of a process. It cannot be changed and it does not depend on which of its ancestors are alive at any given time. The genetic footprint of a process defines which task it belongs to. Genealogy is a good way to define a task for the following reasons. First, resources must be allocated and accounted for during the execution of a job. These resources include, among others, memory, disk space, and processor time. Any process associated with that job should be considered as using those resources. Second, it defines a scheduling unit that can be controlled through various priority schemes. Even though the set of processes comprising a task is dynamically changing, the binding of processes to tasks is established at process creation time and remains fixed until process termination.

We define five mechanisms that are necessary for implementing the genealogy concept in a job management system. For future reference, we name these mechanisms M_1 though M_5:

- M_1: a mechanism to create new tasks,
- M_2: a mechanism to associate processes to tasks,
- M_3: a mechanism to terminate all processes in a task,
- M_4: a mechanism to capture all dynamic process creation and termination in order to establish the genetic footprint, and
- M_5: a mechanism to prevent any "escape hatches" that allow processes to leave a task.

These are in addition to the basic functionality of suspending and resuming execution of a task, necessary for the actual scheduling operation.

In this paper we describe the difficulties that we encountered in implementing the genealogy concept for task control, and what solutions we adopted. In Section 2 we explain why the existing UNIX concepts for process control are not appropriate for the kind of task control we desire. In Section 3 we discuss the particulars of our implementations and in Section 4 we present some experimental results for our task control

mechanism on a single node. Section 5 discusses the integration of process tracking on an actual job scheduling system. Our conclusions are presented in Section 6.

2 Existing UNIX Mechanisms

In standard UNIX systems, processes are the units of scheduling. In order to perform job based scheduling one has to use process set mechanisms. Modern UNIX systems, such as SVR4 and 4.4BSD provide two standard notions of process sets. The first one is *process group* and the second one is *session* [22]. Process groups are provided to identify a set of related processes that should receive a common signal for certain events. Process groups are the standard UNIX mechanism for implementing job control. Sessions are provided to identify a set of related processes that have a common controlling terminal (*i.e.*, are part of one login session). One session can have several process groups.

A process group is defined by its group leader. This is the process which initially creates a new group through a call to *setpgid*. Default process groups are formed by the parent-child relationship that is established when a process forks itself to create another process. The child process is created in the same process group as its parent. Execution of all processes in a process group can be suspended by sending a *signal*(SIGSTOP) to that group. Correspondingly, execution can resume by sending a *signal*(SIGCONT) to that group. At a first glance this seems to implement the genealogy concept previously introduced. However, a UNIX process can switch to a different process group or start its own. This constitutes an escape from its original group, thus failing to implement mechanism M_5 presented above. Also, if a process group leader terminates, all the remaining processes in that group are reassigned to a built-in process group 0. This in turn fails mechanism M_2, since we can no longer associate processes with their task.

Similar to what happens with process groups, a session is defined by its session leader. This is the process which initially creates a new session through a call to *setsid*. Again, default sessions are formed by the parent-child relationship: children initially inherit the session from their parent. Processes can start new sessions, thus disconnecting from their original session and creating the same problems as described for process groups.

In addition to these two explicit ways to define process sets in UNIX (*i.e.*, groups and sessions), there is an implicit way through the transitive parent-child relationship. UNIX provides mechanisms to obtain snapshots (samples) of the list of processes executing at a given time. (Examples of these mechanisms are the *getprocs* function and the *ps* command.) For each process this list reports its parent. By properly processing this list we can build a set that has its origin on a particular root process. However, this fails to establish the proper genealogy of processes for the following reasons. First, it is not an atomic operation: processes can be created and terminated while the list is being examined. Second, if the process list is not sampled at least once between the events of a child being created and its parent terminating, the genetic property of the child is lost. The very first time we come to know about the child process will show *init* as its parent. Finally, since process identifiers are reused in UNIX, one might associate a process with the wrong task if sampling misses this reuse between an old and a new process. In sum-

mary, the parent-child mechanism of UNIX does not properly implement the genealogy concept as required for proper task control.

Some systems have attempted to overcome the shortcomings of the UNIX parent-child model by using a special library that intercepts the *fork()* and *sigaction()* system calls [12]. This approach allows one to record every event of process creation and termination and maintain information on the genealogy of processes. However, this is an incomplete solution, as it depends on the collaboration of user applications in linking to those libraries.

The concept of task control has been successfully implemented in other operating systems. In the OS/390 operating system, the *enclave* is a concept similar to our task [1]. When a request (*e.g.*, web request) enters into the system, and a thread is started in some process to service this request, a new enclave is created. The enclave essentially represents a unit of work being performed on behalf of a request. If this first thread initiates new threads in different processes (*e.g.*, the web request leads to a database operation), then these new threads are also added to the enclave. Accounting of resource consumption is done on an enclave basis. Furthermore, all threads of an enclave can be scheduled concurrently.

3 Implementing Process Tracking

After examining the options offered by UNIX, we did not find any functionality that satisfactorily implements the genealogy concept. It was not an option for us to modify the existing commercial AIX operating system to introduce a new construct. The solution we adopted was the development of a process tracking kernel extension. A kernel extension is a mechanism to dynamically load additional code into the kernel space, thus extending the kernel with new functionality that can be accessed by user-level programs. These extensions execute in the kernel mode of the calling process and have access to all kernel related activities. The purpose of our process tracking kernel extension is to monitor and log all process creation and termination events. Based on that information, the kernel extension maintains the genealogy of selected process sets. The kernel extension also implements the task control functions necessary for scheduling: *task suspend*, *task resume*, and *task kill*.

Our process tracking kernel extension makes use of one particular AIX kernel feature, the process state change handler. As the name indicates, process state change handlers in AIX are invoked every time a process state changes. Several process state change handlers can be chained. In particular, the following events trigger a call to the handlers: process creation, process termination, thread creation, and thread termination. Consequently, process state change handlers have strict performance requirements and have to be as little intrusive as possible. In our case that implies ignoring events unrelated to tracked processes and maintaining the genealogy of tracked processes efficiently.

The operation of the process tracking is illustrated in Figure 2. The kernel extension maintains the following data structures:

– *TrackedProcessObj:* An object of this type is maintained for each process being tracked. It includes the process identifiers (*pid*s) of the process and its genetic par-

ent. It also contains pointers to the *TrackedProcessObj*s for one sibling process and one child process. (The *sibling* pointer is used to form a list of children.) This object is necessary because the kernel process object, the *uproc* structure, cannot be modified directly.

– *TaskObj:* An object of this type is maintained for each task. It contains a task identifier that is assigned by the node level scheduler. It also maintains pointers to the *TrackedProcessObj*s of the top level processes belonging to the task. The collection of all *TaskObj*s constitutes the list of tasks to be monitored in the system.

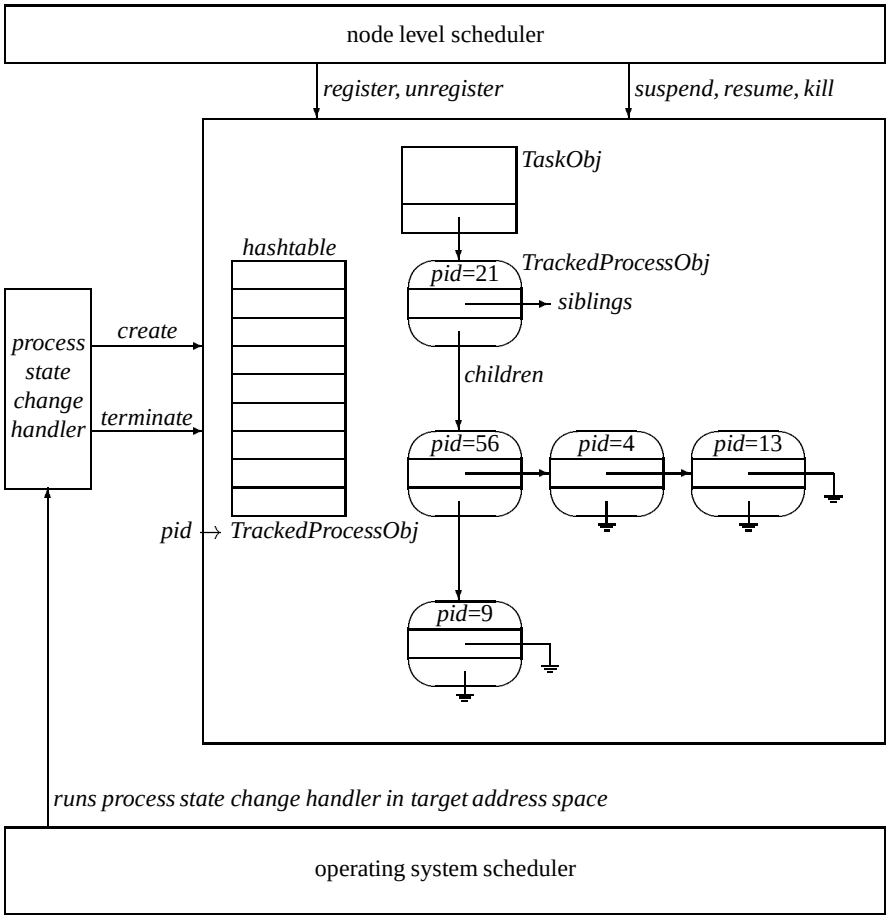


Fig. 2. Organization of the kernel extension.

The process state change handler is loaded once in the kernel extension and immediately starts monitoring process creation and termination events. Because the task list is initially empty, none of these events has any effects. When the node-level scheduler

creates a new task, it registers that task and its root process with the kernel extension. This involves the creation of a new *TaskObj* and a new *TrackedProcessObj*. From this point on the process is being tracked. This implements mechanism M_1. Mechanisms M_2, M_4, and M_5 are implemented by monitoring all process creation and termination events.

When a new process is created, its parent's *pid* is searched in the set of tracked processes. To perform an efficient search we provide a hashtable access based on the *pid*. If the parent is a tracked process, and hence belongs to some task, we create a new *TrackedProcessObj*, add it to the children list of the parent process, and create a new entry in the hashtable.

When a tracked process terminates, we remove its associated *TrackedProcessObj* from its parent's children list. Then its own children are added to its parent's children list. (The *TaskObj* acts as a parent to the top level processes.) This is very different from the way the kernel maintains the parent-child relationship. Whereas the kernel makes *init* adopt orphan processes, we implement a policy in which orphans are adopted by their most immediate ancestor still alive. (As an alternative policy, orphans could always be adopted by the corresponding *TaskObj*.) This approach always keeps a process associated with its original task.

All accesses to the kernel extension functions are serialized via a lock. Through this we implement the atomicity required by the task control operations. Task execution is controlled from the kernel extension through standard UNIX signaling mechanisms as follows.

Task Suspend: Using a top-down depth-first traversal of the task's *TrackedProcessObj* tree, we issue a *signal*(SIGSTOP) to suspend the execution of each process. We verify that the process indeed has stopped before proceeding to the next process. We have to use a traversal order that guarantees that a parent process is stopped before its children are signaled. This avoids the scenario where a parent detects its child stopped and takes some action. (For example, in *csh* this situation implies that *CTRL-Z* was issued to the child process). If a process does not stop immediately, we exit from the suspend operation with an EAGAIN error code. (In a multiprocessor system, a process being stopped could be active on a different processor. In this case, it will only receive the signal when it returns from a system call or when it is about to run in a future OS time-slice.) Upon detecting the EAGAIN error code, the node-level scheduler retries the operation after waiting for a period we refer to as the *retry interval*. During the subsequent invocation, the *TaskSuspend* function verifies whether a process has already been signaled previously with the same signal and whether it has indeed stopped. If so, we continue signaling the remaining processes. This mechanism deals properly with the dynamics of process creation and termination between retries. (Note that, on a retry, we do not have to resend signals, just make sure that they have taken effect.)

Task Resume: Using a bottom-up depth-first traversal of the task's *TrackedProcessObj* tree, we use *signal*(SIGCONT) to resume execution of each process. We verify that the process has indeed resumed execution before proceeding to the next process. This traversal order is the opposite of what we use when suspending a task. Again, the goal is to avoid the scenario where a parent detects its child stopped. If a process does not

resume immediately, we exit from the resume operation with an EAGAIN error code, to let the node level scheduler retry this operation.

Task Kill: The algorithm to kill a task also uses a top-down traversal of the task's *TrackedProcessObj* tree, issuing a *signal*(SIGKILL) to each process. Since the action of a SIGKILL is guaranteed, there is no need to test before proceeding to the next process. This is the implementation of mechanism M_3.

Task suspend, resume, and kill operations can also be selectively performed on a subtree of the processes of a task.

An odd situation can arise due to the access serialization in the kernel extension functions. When a process in the middle of its creation event recognizes that its parent has stopped, it is immediately stopped as well. This mechanism, together with the top-down traversal, guarantees that, with a finite number of retries, progress is made in suspending the task. Ultimately, the number of processes that can be created is limited by the operating system. At one point, all processes belonging to the task will be suspended.

4 Experimental Results

In this section we discuss an experimental evaluation of our task control mechanism. We conduct this evaluation through direct measurement of the context switch time between two tasks. This is a very common operation performed by the process tracking facility when implementing time-sharing. A complete context switch operation, from a currently running task A to a currently suspended task B, requires first suspending task A and then resuming task B. The mechanism for suspending and resuming tasks is discussed in Section 3.

For the purpose of our experiments we use tasks that are binary trees of processes. We have chosen three types of tasks: 2, 3, and 4 levels deep, for a total of 3, 7, and 15 processes per task, respectively. The tasks use very little memory and all fit comfortably within the physical memory of our test systems. We measure the context switch time between two tasks with the same number of processes. The measurements are repeated several times and the results presented in this paper represent arithmetic means of the samples. We perform the measurements for different values of the retry interval, that is the interval the node level scheduler waits before retrying a suspend or resume operation that fails.

The first set of results that we present is for a uniprocessor machine: an 160 MHz P2SC thin-node of an IBM RS/6000 SP system, with 256 MB of main memory. Results for this machine are shown in Figure 3. The context switch operations are very fast, taking less than 1 ms to complete. Also, the context switch time is proportional to the number of processes in the tasks, as expected from the description in Section 3. Finally, the context switch time is invariant to the retry interval. This occurs because retries are extremely infrequent. A suspended process cannot be running, therefore a *signal*(SIGCONT) to such a process has immediate effect, bringing it to active state.

Correspondingly, if the node level scheduler is running and performing a suspend operation, the processes from the tasks being suspended are not running (this is a uniprocessor system). Hence, the *signal*(SIGSTOP) takes effect immediately and the processes are suspended. (There are some rare situations when the signal cannot be processed immediately and therefore retries are necessary.)

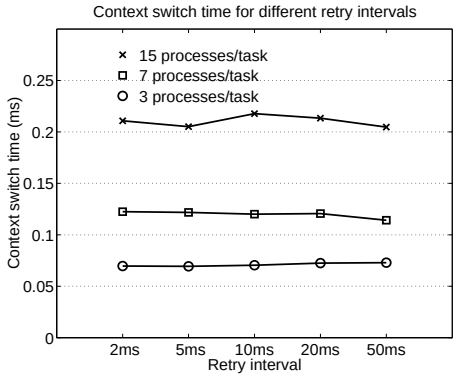
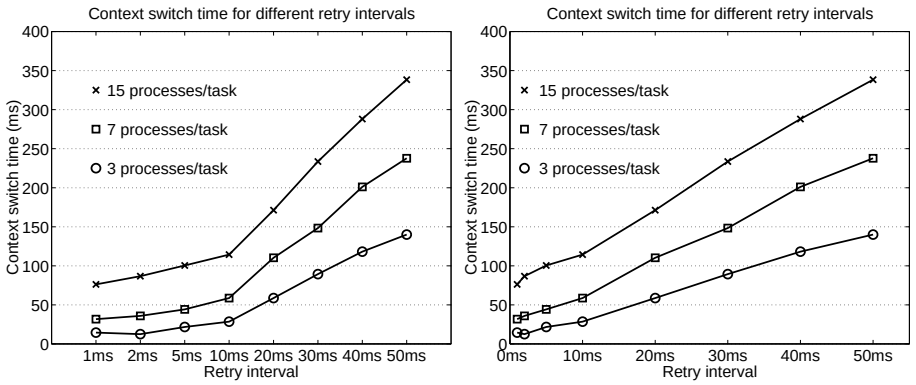


Fig. 3. Context switch times for a uniprocessor system.

We repeat these experiments on a multiprocessor machine: an IBM RS/6000 model S70 system. This machine has four 125 MHz 64-bit RS64 processors and 512 MB of main memory. Results for this machine are shown in Figure 4. We present the same data using nonlinear (Figure 4(a)) and linear (Figure 4(b)) time scales. (The nonlinear scale shows more detail in the 1-10 ms range, while the linear scale allows for a better understanding of the behavior in the 10-50 ms range.) The major difference in comparison to Figure 3 is the effect of the retry interval on the total context switch time. Overall, the context switch time is much larger than on an uniprocessor, and it increases with the retry interval. We also observe, in Figure 4(b), that the context switch time is approximately linear on the retry interval in the 10 to 50 ms range. Context switch times on a multiprocessor are dominated by the time to suspend a running task. Resuming a suspended task still does not require any retries and is very fast. The maximum value we observed for resuming a task with 15 processes was approximately 0.3ms.

The interesting behavior occurs when suspending a running task. Because we are now dealing with a multiprocessor system, it is possible for an application process to be executing at the same time the node level scheduler is trying to suspend it. When that occurs, a *signal*(SIGSTOP) is sent to the process but it is not actually processed until the next time the process is about to run. The suspend operation fails and the node level scheduler has to retry.

We can model the context switch time as a function of the retry interval as follows. Consider a multiprocessor system with n processors and only the following processes: the node level scheduler, the processes from the active task and the processes from the suspended task. We know that the context switch time is dominated by the time



(a) nonlinear time scale (b) linear time scale

Fig. 4. Context switch times for a multiprocessor system.

to suspend processes that are actually running. Hence, that will be the focus of our following discussion. Let τ be the retry interval used by the node level scheduler and let there be p processes in the task to be suspended. Let t_k be the time that it takes to suspend one process when there are still k processes active. The total time θ to suspend the active task is:

$$\theta = \sum_{k=1}^{p} t_k . \tag{1}$$

Time t_k can be computed as

$$t_k = P_r(k, n) \times \overline{r} \times \tau \tag{2}$$

Where $P_r(k, n)$ is the probability that the particular process to be suspended is running on one of n processors and $\overline{r}$ is the average number of retries before it is stopped. (If the process is not running, the time to stop it is negligible, as seen for the uniprocessor system.) We know that the node level scheduler is running on a processor for sure, leaving $n - 1$ processors for the task processes. Also, if $k \leq n - 1$, then the process must be running on some processor. Therefore, we can write

$$P_r(k, n) = \min(1, \frac{n-1}{k}). \tag{3}$$

The mean number of retries to stop a running process is

$$\overline{r} = \left\lceil \frac{T_{\text{stop}}(k, n)}{\tau} \right\rceil . \tag{4}$$

$T_{\text{stop}}(k, n)$ is the time it takes for a running process to respond to *signal*(SIGSTOP) when there are k processes active on n processors. (The node level scheduler sleeps between retries, so all n processors are available for running task processes.) At each system clock tick of length T, a system with n processors runs $\min(k, n)$ processes. (In

AIX, this clock tick is typically 10 ms.) Because of the round-robin policy for processes at the same priority level, it will take

$$\frac{k}{\min(k,n)} - \frac{1}{2} \tag{5}$$

clock ticks for the signaled process to be scheduled again. (The $-1/2$ term accounts for the fact that a *signal*(SIGSTOP) can be issued any time during the current clock tick.) The actual time for stopping a running process is:

$$\begin{aligned} T_{\text{stop}}(k,n) &= \left(\frac{k}{\min(k,n)} - \frac{1}{2}\right) T \\ &= \left(\frac{2k - \min(k,n)}{2\min(k,n)}\right) T \end{aligned} \tag{6}$$

and

$$\overline{r} = \left\lceil \frac{(2k - \min(k,n))T}{2\min(k,n)\tau} \right\rceil \tag{7}$$

Substituting Equation (3) and Equation (7) into Equation (2) leads to

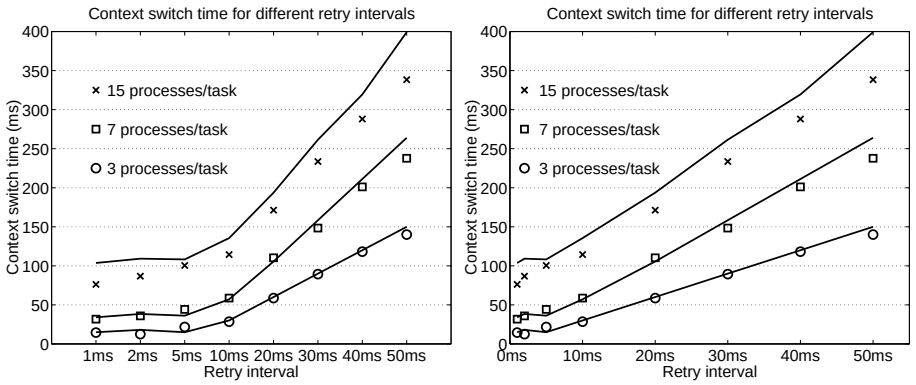
$$t_k = \min(1, \frac{n-1}{k}) \left\lceil \frac{(2k - \min(k,n))T}{2\min(k,n)\tau} \right\rceil \tau \tag{8}$$

and

$$\theta = \sum_{k=1}^{p} \min(1, \frac{n-1}{k}) \left\lceil \frac{(2k - \min(k,n))T}{2\min(k,n)\tau} \right\rceil \tau. \tag{9}$$

The correspondence between our model and experimental results is shown in Figure 5, again with two different time scales. Values from the model are shown in solid lines, whereas the markers represent the experimental data. Overall agreement is very good, with a tendency by the model to overestimate the context switch time. This tendency might be explained by the presence of other processes in the system. In real UNIX systems there are many active processes at any given time. For example, even in single-user mode our multiprocessor system had 70 active processes, in addition to the task processes of our experiments. These are usually system daemons that consume little resources, maybe 1 to 2% of total CPU time when combined. Nevertheless, they contribute to decrease the probability that application processes will be running at any given time, thus reducing the total suspend time for a task.

Both the model and the experimental results show that there is little variation in the context switch time when the retry interval is varied between 1 and 5 ms. Retrying too soon is ineffective, as the process being signaled will not have had time to process the signal. Since each retry operation is a kernel extension invocation involving kernel locks, repeating it too often can hurt system performance. Therefore, a retry interval of 5 or 10 ms is the best choice for the node-level scheduler on this system.



(a) nonlinear time scale (b) linear time scale

Fig. 5. Model and experimental data for context switch times for a multiprocessor system.

5 System Integration

Process tracking is currently used to implement task control in a research prototype gang-scheduling system for the RS/6000 SP [15]. We discuss this system here as an example of an application of process tracking. We emphasize that process tracking is applicable in many other environments. The general problem that process tracking tackles is that of resource reclaiming: We want to be able to get back resources that a job is using in order to give them to another job. In the case of gang-scheduling, that resource is (mainly) processor time. Process tracking can also be used to control "stray" processes that would typically overlast the lifetime of a job and continue to consume system resources.

Gang-scheduling is a mechanism for performing coordinated scheduling of the tasks of a parallel job [16]. It partitions the resources of a parallel system both in space and time, and all tasks of one job execute concurrently. Gang-scheduling has been shown to improve system performance, in particular by improving system utilization and reducing job wait time [5, 15]. The operation of a gang-scheduled system is characterized by a stream of context-switch events, that occur at the boundaries of the time-slices. On each context-switch event a currently running parallel job is suspended and another job is enabled for execution.

The gang-scheduling system we developed follows the two-tier model discussed in Section 1, with a centralized global scheduler and local node-level schedulers. The role of the central scheduler is to derive a global schedule for the system and then distribute the relevant subsets to the local schedulers in an efficient manner. While there are many approaches to representing global schedules, we have chosen the Ousterhout matrix, due to its simplicity and generality. Using hierarchical distribution schemes, we deliver each column of the matrix to its designated node-level scheduler. For multiprocessor nodes, the corresponding node-level scheduler receives multiple columns. The node-level scheduler is then responsible for implementing this local schedule as described by the columns of the Ousterhout matrix. It accomplishes that by executing the con-

text switch operations at the time dictated by the schedule. The system relies on some form of a synchronized clock for all node-level schedulers. This can be either a global clock, or distributed local clocks that are kept synchronized with NTP [14]. Note, that in the case of a multiprocessor node, a single node-level scheduler may have to perform multiple context switch operations.

The environment we use to measure application performance under our gang-scheduling system consists of an IBM RS/6000 SP with 9 compute nodes and an additional node dedicated to handling job submission and running the global scheduler. Each compute node has four 332 MHz PowerPC 604e processors that share 1.5 GB of main memory. Job execution is controlled by our gang-scheduling prototype, using a time-slice of 10 seconds. As benchmarks we use the three pseudo-applications from the NAS Parallel Benchmark suite, version 2.3 [2]: BT, LU, and SP. Each benchmark is written in Fortran with calls to the MPI message-passing library. BT and SP are compiled to run with 36 tasks (requiring all 9 nodes), while LU is compiled to run with 32 tasks (requiring 8 nodes). Each task consists of exactly three processes. One of the processes implements the benchmark itself, while the other two are support processes that implement the parallel environment.

We first run each benchmark on a dedicated environment and measure their execution time. This corresponds to gang-scheduling with a multiprogramming level (MPL) of one. We then run two, three, and four instances of each benchmark at a time, which corresponds to MPLs of 2, 3, and 4, respectively. Results from these experiments are shown in Table 1 and in Figure 6. For each benchmark, Table 1 shows the memory footprint *per task*, and the average execution time (in seconds) under different multiprogramming levels. Figure 6 presents the same performance results graphically, with the execution times for each benchmark normalized to the execution time of a single instance of the benchmark in a dedicated environment, and then divided by the multiprogramming level. We note that the highest memory consumption occurs for BT with an MPL of 4. In that case, there are 16 tasks running in each node, for a total memory footprint of 590 MB. (The support processes have small footprints.) This is still much smaller than the 1.5GB of main memory available in each node, and therefore there is minimal paging during execution of the benchmarks.

Table 1. Results for running the NAS Parallel Benchmarks BT, LU, and SP under gang-scheduling.

benchmark	number of tasks	footprint (MBytes)	Average execution time (s)			
			MPL = 1	MPL = 2	MPL = 3	MPL = 4
BT, class B	36	36.9	568.79	1157.14	1763.32	2339.36
LU, class B	32	8.5	371.21	743.92	1202.03	1599.72
SP, class B	36	15.2	432.95	916.27	1360.56	1842.16

In the ideal case, running k instances of a benchmark should slow their execution by a factor of k. In this ideal case, all bars in Figure 6 should be at a value of 1. Any

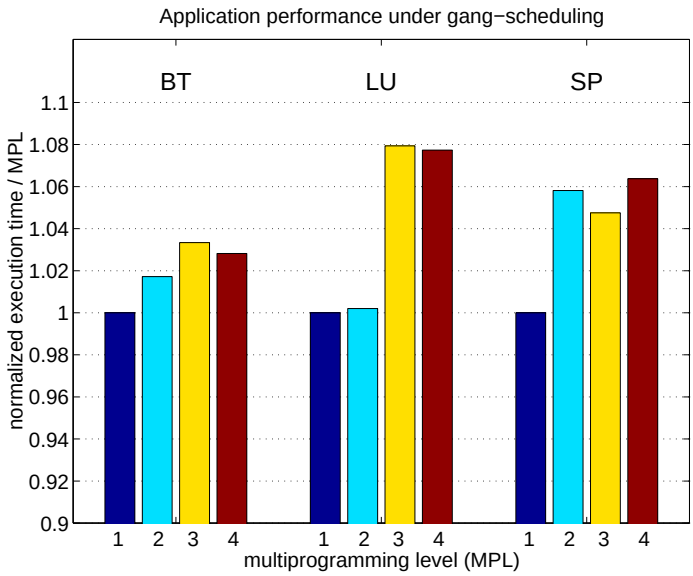


Fig. 6. Performance of BT, LU, and SP under gang-scheduling.

value above 1 represents overhead resulting from time-sharing. We note that the worst overhead occurs for LU with multiprogramming levels of 3 and 4. In those cases, the measured execution time is 8% larger than expected from the ideal case. Overall, our gang-scheduling system does an effective job of providing the illusion of a slower virtual machine for the execution of each job. Explicit control of the time slices allows us to amortize the cost of context switching over a long enough time slice. It also allows for resource dedication for longer periods and thus one can expect higher cache hit rates, lower page fault rates, and better communication performance (due to the synchronous nature of the applications). The footprint of our benchmarks were too small to exercise the memory paging system, and we plan to conduct such experiments in the near future.

6 Conclusions and Future Work

In this paper we introduced the new concept of process genealogy to define the set of processes comprising a task. Process to task binding is an intrinsic characteristic of the process which it inherits from its parent and which can not be modified. None of the existing process set concepts in UNIX satisfy the requirements for building task based genealogy. We have successfully implemented the genealogy mechanisms without modifying the operating system. Our approach is based on a process tracking kernel extension that monitors process creation and termination events and builds a database representing the genealogy concept. Process tracking also implements atomic suspend and resume operations for tasks. These operations form the core of context-switching for time-sharing systems.

We have used process tracking as an integral part of our gang-scheduling system for the RS/6000 SP. An initial prototype of this system has been installed at Lawrence Livermore National Laboratory. Deployment in production mode will follow soon. Our preliminary results indicate little overhead from the context-switching as performed by the process tracking facility. As an added benefit, we eliminated the possibility of any stray processes escaping termination control. The development of process tracking was motivated by the necessary functionality. Nevertheless, our experiments have shown that we can use it to implement efficient, low overhead task control. (The experiments have also shown the importance of using the proper retry interval, in the 5 to 10 ms range, in suspending tasks. 10 ms is the base operating system time-slice.) Process tracking is a vehicle through which an enhanced user-level scheduling can be added to a system.

In terms of future work, we are investigating alternative ways to traverse the forest of processes that constitute a task. We currently perform a depth first traversal, which we quit (and later retry) when we find a process that is not immediately suspended or resumed. We could also use a breadth first traversal: Attempt to suspend/resume all processes at a given level before proceeding to the next level. This approach would require changes in our data structures and needs to be investigated. There is also a hybrid, or optimized, depth first approach: We start traversing the tree in depth first mode. If we hit a process that we cannot stop then, instead of proceeding to its children, we just move on to its siblings. (See Figure 7.) This traversal can be implemented with our current data structures.

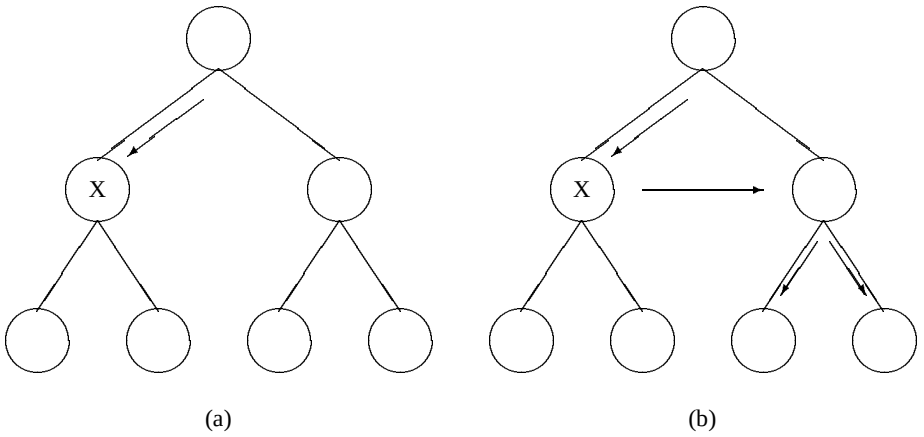


Fig. 7. Depth first approach (a) stops as soon as one task (marked X) cannot be suspended or resumed. The hybrid approach (b) continues with the siblings of task X.

Process genealogy is an important concept for job control and accounting. We have successfully implemented the genealogy concept through process tracking in a commercial operating system. Other implementation strategies are also possible. In particular,

as a new research, we are currently investigating the migration of the genealogy concept directly into the kernel of the AIX UNIX operating system.

Acknowledgements: The authors wish to thank Marc Snir and Richard Lawrence of IBM Research for supporting our work, and Morris Jette and Andy Yoo of Lawrence Livermore National Laboratory for useful discussions during the design of the system and help with the experiments. Special thanks to our IBM colleagues Liana Fong and Waiman Chan, who worked with us in the design and implementation of the gang-scheduling system.

References

1. J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. **Adaptive Algorithms for Managing a Distributed Data Processing Workload**. *IBM Systems Journal*, 36(2):242–283, 1997.
2. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. **The NAS Parallel Benchmarks 2.0**. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
3. A. C. Dusseau, R. H. Arpaci, and D. E. Culler. **Effective Distributed Scheduling of Parallel Workloads**. In *ACM SIGMETRICS'96 Conference on the Measurement and Modeling of Computer Society*, 1996.
4. D. G. Feitelson. **A Survey of Scheduling in Multiprogrammed Parallel Systems**. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.
5. D. G. Feitelson and M. A. Jette. **Improved Utilization and Responsiveness with Gang Scheduling**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, April 1997.
6. D. G. Feitelson and L. Rudolph. **Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control**. *Journal of Parallel and Distributed Computing*, 35:18–34, 1996.
7. H. Franke, P. Pattnaik, and L. Rudolph. **Gang Scheduling for Highly Efficient Multiprocessors**. In *Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Maryland*, 1996.
8. B. Gorda and R. Wolski. **Time Sharing Massively Parallel Machines**. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.
9. N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. **Extensible Resource Management for Cluster Computing**. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 561–568, 1997.
10. M. A. Jette. **Expanding Symmetric Multiprocessor Capability Through Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.
11. D. Lifka. **The ANL/IBM SP scheduling system**. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer-Verlag, April 1995.
12. M. J. Litzkow, M. Livny, and M. W. Mutka. **Condor – A Hunter of Idle Workstations**. In *Proceedings of 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, 1988. Project URL: `http://www.cs.wisc.edu/condor/`.
13. Maui High Performance Computing Center. **Maui Scheduler Administrator's Manual**, 1998. URL: `http://wailea.mhpcc.edu/maui/doc/`.

14. D. Mills. **Network Time Protocol (Version 3) Specification, Implementation and Analysis**. Technical Report RFC 1305, University of Delaware, March 1992.
15. J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette. **An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments**. In *Proceedings of SC98, Orlando, FL*, November 1998.
16. J. K. Ousterhout. **Scheduling Techniques for Concurrent Systems**. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.
17. U. Schwiegelshohn and R. Yahyapour. **Improving First-Come-First-Serve Job Scheduling by Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.
18. J. Skovira, W. Chan, H. Zhou, and D. Lifka. **The EASY-LoadLeveler API project**. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, April 1996.
19. P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. **Dynamic Coscheduling on Workstation Clusters**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.
20. P. G. Sobalvarro and W. E. Weihl. **Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors**. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 106–126. Springer-Verlag, April 1995.
21. K. Suzaki and D. Walsh. **Implementation of the Combination of Time Sharing and Space Sharing on AP/Linux**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.
22. U. Vahalia. **UNIX Internals: The New Frontiers**. Prentice-Hall, Inc, 1996.