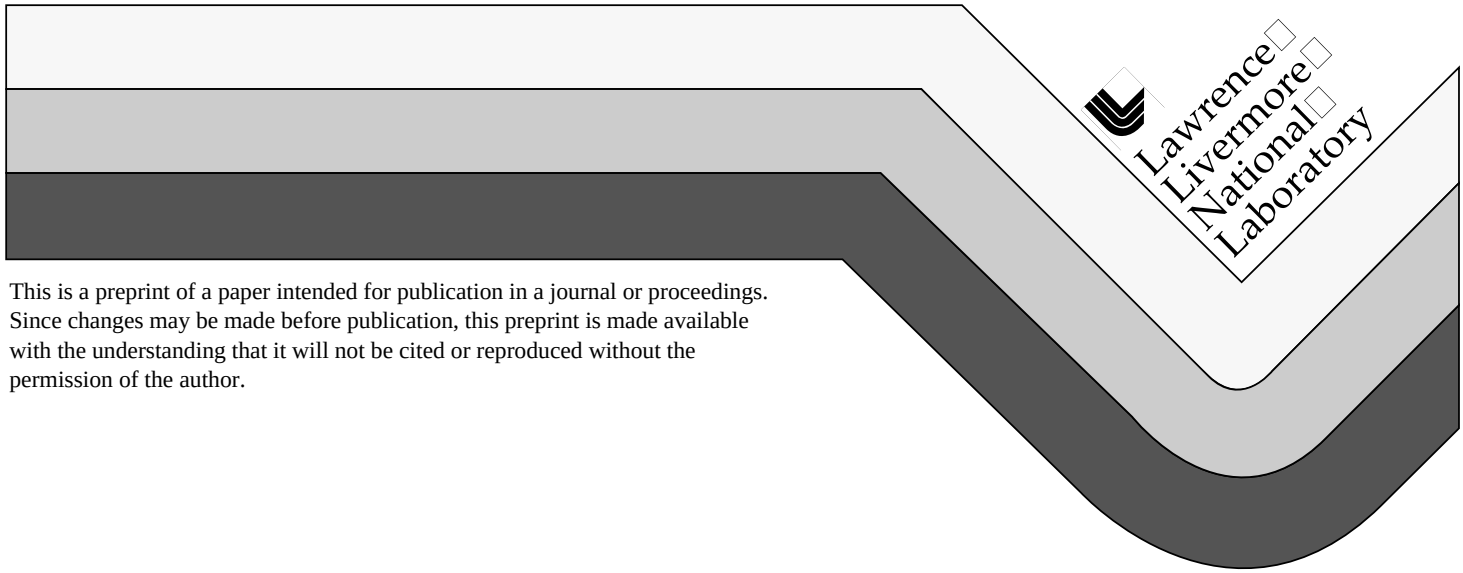
UCRL-JC-129927
PREPRINT

Expanding Symmetric Multiprocessor Capability Through Gang Scheduling

M.A. Jette

This paper was prepared for submittal to the
International Parallel Processing Symposium
Job Scheduling Strategies Workshop
Orlando, FL
March 30-April 3, 1998

March 1998



Expanding Symmetric Multiprocessor Capability Through Gang Scheduling

Morris A. Jette
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550
jette@llnl.gov
http://www-lc.llnl.gov/global_access/dctg/jette/

Abstract:

Symmetric Multiprocessor (SMP) systems normally provide both space-sharing and time-sharing to insure high system utilization and good responsiveness. However the prevailing lack of concurrent scheduling for parallel programs precludes SMP use in addressing many large-scale problems. Tightly synchronized communications are impractical and normal time-sharing reduces the benefit of cache memory. Evidence gathered at Lawrence Livermore National Laboratory (LLNL) indicates that gang scheduling can increase the capability of SMP systems and parallel program performance without adverse impact upon system utilization or responsiveness.

Introduction

Parallel computer systems have been in use at LLNL since the introduction of a 126 processor BBN TC2000 computer in 1989. Subsequent deployments of Meiko CS-2, Cray C90, Cray T3D, IBM SP, and Digital Alpha systems have encouraged parallel application program development. The majority of LLNL's workload consists of numerical analysis programs designed for 16 to 256 way parallelism with memory requirements in excess of one gigabyte, disk space requirements in the 10 to 10000 gigabyte range, and execution times in the 1 to 40 hour range.

While Massively Parallel Processing (MPP) systems are well suited for execution of existing programs, the scheduling mechanisms available on some systems make program development somewhat difficult. Once a parallel program on the Meiko CS-2 or IBM SP begins execution, processors are dedicated to the program until its termination. Multiple parallel programs may execute concurrently on distinct processors, but will not execute simultaneously on any processor. In order to provide good responsiveness for program development at LLNL, small numbers of processors are placed in a partition available only to programs with short execution times and small processor counts. Larger programs may experience delays of many hours in order to execute outside of program development partitions.

SMP systems normally have multiple processors sharing a common workload and memory. Distinct programs may execute on each processor and a program's threads of execution may migrate between processors to provide good responsiveness and high system utilization. Many of our customers find the programming environment on Digital Alpha computers to be particularly appealing with a large memory space shared by eight to 12 processors. The Digital Alpha processor performance is also excellent and attracts interest for execution of moderate size problems. Some applications require faster throughput than can be provided by a single processor and utilize multitasking to achieve this. Multitasking a program can provide some performance enhancement, but performance can vary widely with system

load.

There are some UNIX scheduler implementation differences, but most systems maintain one or more queues of runnable threads [1, 2]. Whenever a processor becomes available, the highest priority thread is selected to execute. The thread's priority may be based upon a history of recent processor utilization, reason for last relinquishing a processor (eg. waiting for I/O completion), process nice value, and priority class (real-time or time-sharing). The thread continues execution for some time quanta which is dependent upon the process priority and priority class. This algorithm tends to maximize system utilization and responsiveness. In most cases, no effort is made to concurrently schedule the threads which comprise a single parallel program. On a computer without concurrent scheduling and more runnable threads than processors, the components of a parallel program may experience synchronization delays due to poor overlap in scheduling. Many tightly synchronized programs continuously poll semaphores at synchronization points (spin-wait) rather than relinquishing the processor. Unless the program's threads of execution can be provided with processors in a synchronized fashion, this spin-wait time can consume substantial resources without advancing the application's progress.

Figure 1 shows the behavior which might be experienced by a six thread parallel program executing on an eight processor multiprogrammed computer without gang scheduling. Processor use for only a portion of the parallel program's execution time is shown. Other running programs consume the remainder of processor resources and this is not shown. Most SMP systems fail to provide synchronized compute resources for parallel programs and even slight levels of competition for processors can severely impact the program's performance. The problem is most severe for programs with large thread counts on heavily utilized systems.

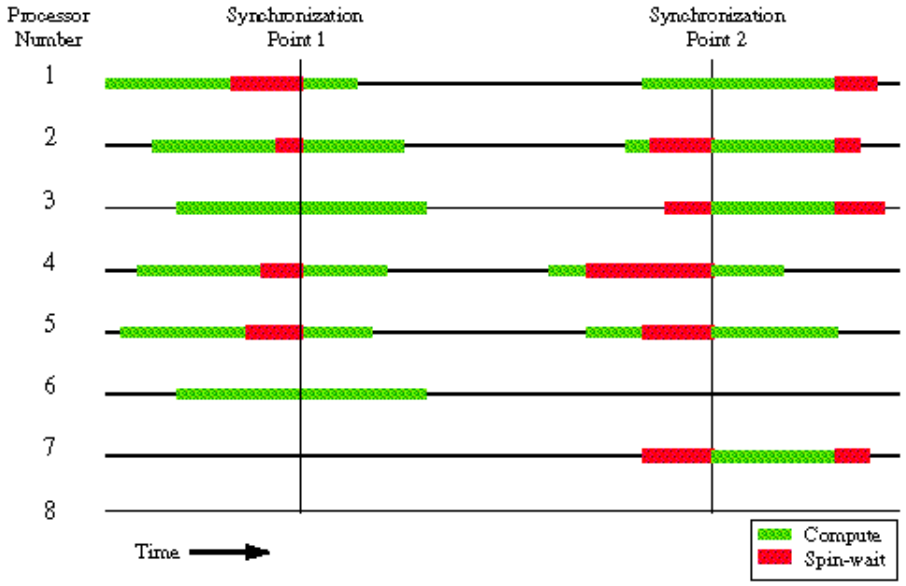


Figure 1: Parallel program performance without gang scheduling

Gang scheduling groups a program's parallel threads of execution into a gang, then concurrently schedules an independent processor to each thread in the gang [5]. A thread here is broadly defined as being a path of program execution which can proceed concurrently with others. Included in this definition are processes generated by fork system calls, MPI (Message Passing Interface) and PVM

(Parallel Virtual Machine) programs, as well as Pthreads. MPI and PVM threads may span multiple computers. Multiple programs may execute independently on distinct processors at the same time, referred to as *space-sharing*. Time-sharing is supported by providing the gang scheduled program access to processors as well as removing that access concurrently. Time-sharing is used to prevent starvation of any program or achieve other resource distribution criterion. The gang scheduled program is provided with the perspective of dedicated resources during its periods of execution, with the exception of memory and I/O bandwidth. Figure 2 shows the dramatic reduction in spin-wait overhead which the sample program might experience with gang scheduling on the Digital Alpha. While perfect synchronization can not be provided with the Digital UNIX system's infrastructure, it can provide quite good synchronization as explored later in the paper. The program's throughput can be significantly improved by reducing spin-wait time without significant impact upon either overall system throughput or responsiveness. Gang scheduling is one of those rare circumstances when it is possible to get something for (almost) nothing.

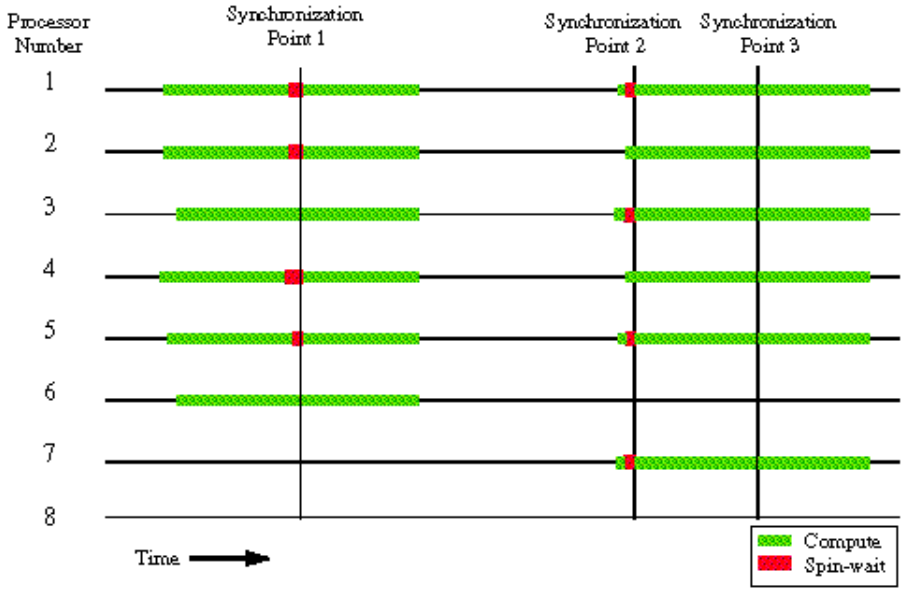


Figure 2: Parallel program performance with gang scheduling

Several studies of scheduling algorithms indicate that gang scheduling is a relatively good policy [5, 9, 14]. Gang schedulers have been implemented on a variety of computer platforms including Cray T3D [6], Cray T3E [12], CM-5, and Silicon Graphics multiprocessor workstations [3]. This paper describes a gang scheduler implementation for Digital computer systems and its performance characteristics both on a single computer and across a cluster.

Digital UNIX

LLNL has two clusters of Digital Alpha 8400 computers. The cluster for unclassified work includes eight computers with a total of 80 440 MHz processors, 56 gigabytes of memory, and 800 gigabytes of local disk. These computers are interconnected with a Digital *memory channel* with performance that permits high-performance problems to effectively span a cluster. The memory channel has a latency of 3 microseconds and bandwidth in excess of 100 megabytes per second. This compares with 0.5

microsecond latency and bandwidth in excess of 500 megabytes per second for the computer's bus. A comparably sized cluster exists exclusively for classified work.

The Digital UNIX 4.0D operating system includes a very fine grained fair-share scheduler called a *class scheduler*. Each process and its threads may be associated with a specific class and each class has a target resource allocation. If class scheduling is configured, each process is by default associated with the class *default*. For example, consider an eight processor system with two classes defined: *default* and *gang.job.1*. One might associate a four thread program with *gang.job.1* and target the class at 50 percent of resources to provide it with four processors. The remaining 50 percent of resources, or four processors, would be available to processes in the *default* class and managed through normal UNIX scheduling.

Modifications to the class scheduler database are performed via an Application Program Interface (API) to a class scheduler daemon. The class scheduler daemon's database is propagated to the operating system kernel immediately when a process is added or removed from a class. Changes in a class' target resource allocations are propagated to the kernel at configurable intervals of one second or longer. The kernel then maintains precise resource utilization statistics for each class. These statistics are used in conjunction with normal UNIX process scheduling priority to assign a runnable thread to a processor available for scheduling. Threads belonging to classes exceeding their target resource allocation will either be scheduled only to prevent a processor from becoming idle or will be completely prevented from executing, depending upon a configurable parameter. Executing threads are not preempted prior to completion of their normal time quanta nor are classes assured of achieving their target resource allocation on a short-term basis, so a gang scheduler's ability to concurrently schedule threads is imperfect.

While the class scheduler infrastructure may be less than ideal for implementing a gang scheduler, it does offer some interesting capabilities. If a parallel program is unable to fully utilize its target resource allocation, those resources (processors) are automatically reallocated to other programs in order to sustain high overall system utilization. This minimizes the negative impact of unbalanced parallel applications and those with significant I/O components. Processes run by user *root* are exempt from class scheduling constraints, which insures that system functionality will be maintained at a cost of reduced control for the gang scheduler. Since changes in a class' target resource allocation require on the order of one second to propagate into the kernel, gang scheduling with this mechanism necessitates time-slice durations at least this large to be effective.

Class scheduling makes no attempt to bind specific processors to specific threads. Digital UNIX does calculate the highest priority thread for each available processor and the last processor used by each thread is a factor in this calculation. This algorithm limits movement of threads between processors and reduces the overhead of refreshing a processor's cache. The overall rate of context switches for a parallel program on a heavily utilized computer is reduced by about 50 percent with this gang scheduler compared with normal Digital UNIX scheduling. Thread migration between processors is reduced by a similar amount. The binding of threads to processors may very well reduce cache refresh overhead, but at a cost of reduced processor scheduling flexibility. Investigation of this issue has been deferred.

Table 1 illustrates the speedup actually achieved by a gang scheduled compute-bound benchmark on a multiprogrammed computer. Efficiency here is defined as the speedup divided by the benchmark's thread count. This twelve processor system provided excellent speedup despite interference from about twenty other runnable threads throughout the testing period (the computer was in normal production use

at this time with a heavy interactive load). Near perfect efficiency was achieved at low levels of parallelism. High levels of parallelism experienced less efficiency and greater variation in results, apparently due to difficulties faced by the class scheduler in managing far more runnable threads than processors.

Thread Count:	1	2	3	4	5	6	7	8	9	10	11
Speedup:	1.000	1.983	2.998	3.989	4.992	5.968	6.928	7.876	8.791	9.665	10.511
Efficiency (Percent):	100.0	99.1	99.9	99.7	99.8	99.5	99.0	98.4	97.7	96.6	95.6

Table 1: Speedup achieved with gang scheduling on a busy computer

Gang Scheduler Design

The gang scheduler developed by LLNL for Digital clusters is an evolution of earlier ones developed for the BBN TC2000 [7, 8] and Cray T3D [6, 10, 11] systems. Both implementations were very successful at adding a time-sharing capability to these MPP systems, which otherwise provided both space-sharing and concurrent scheduling of resources. The Cray T3D was able to sustain weekly CPU utilization over 96 percent while the aggregate interactive workload slowdown was only 18 percent (amount by which elapsed time exceeded run time). One important feature of this design is the classification of each program in terms of scheduling requirements. The following prioritized job classes are supported in the Digital implementation:

- **Express jobs** are deemed by management to be mission critical and are given rapid response and optimal throughput. Programs may be placed into the express class only by system administrators.
- **Interactive jobs** require rapid response time and very good throughput during working hours. The response time and throughput may be reduced at other times for the sake of improved system utilization or throughput of batch jobs.
- **Batch jobs** do not require rapid response, but should receive very good throughput outside of working hours.
- **Standby jobs** have low priority and are suitable for absorbing otherwise idle compute resources. Programs are normally placed into the standby class after the user or his group have consumed more resources than desired by management.

Users may submit programs to the interactive, batch, and standby classes. The class of a program may be altered to a lower priority class by the user at any time. The system administrator may set any program to any job class.

The implementations for BBN and Cray systems were able to take advantage of vendor supplied parallel job initiation software to perform gang scheduling without application modification. The Digital environment lacks a single parallel job initiation mechanism, making the application interface more complex. At least four distinct parallel job initiation mechanisms exist: MPI, PVM, Pthreads, and fork calls. These mechanisms are utilized through compilers, libraries, and/or explicit user request. It is also common to combine multiple mechanisms in a single program, such as a PVM program spanning multiple computers but using Pthreads within each computer for improved performance.

For the Digital gang scheduler implementation, minor application or library changes were deemed necessary to register each program and process to be gang scheduled. These functions are provided through an API which issues Remote Procedure Calls (RPC) to one or more the gang scheduler daemons. The program registration function includes the job class and for each computer to be used: desired processor count, minimum processor count, desired real memory space, and desired disk space. This RPC contacts the gang scheduler daemon on each computer to be used and returns a single global job ID. For process registration, each process ID to be associated with a global job ID is specified. These calls were embedded into LLNL's version of the MPICH library and automate gang scheduling for users of that library with the setting of an environment variable. Other programs must have the necessary modifications made directly to the code, typically 20 to 50 lines of code. While this entails some programming effort, it can function with any combination of programming models and communications mechanisms within a computer or across multiple computers. A simple example of program and process registration is shown in the appendix. Additional API calls can be used to dissociate a process from a program, changes a program's class, modify resource requirements, gather resource utilization information, and query a computer's load. API functions are provided for both C and FORTRAN programs, which each account for roughly half of our workload.

The API writes the program's request into a file of a global file system and communicates with the gang scheduler daemons using sockets and a well known port. The RPC contains user identification and the file's location. Daemons receive the RPC, confirm the file's ownership for authentication, perform the requested action, and reply over the socket. This mechanism provides good security, flexibility, and performance.

Program and computer status information is written to a globally readable file at the start of each gang scheduler time-slice. An x-window program, *xgang*, reads this file and reports computer and program status as shown in figure 3. Limited program modification capabilities are also provided by xgang. A user may modify a program's class, suspend, resume, or kill it across all computers with the push of a button. xgang has also proven quite useful for monitoring overall system performance.

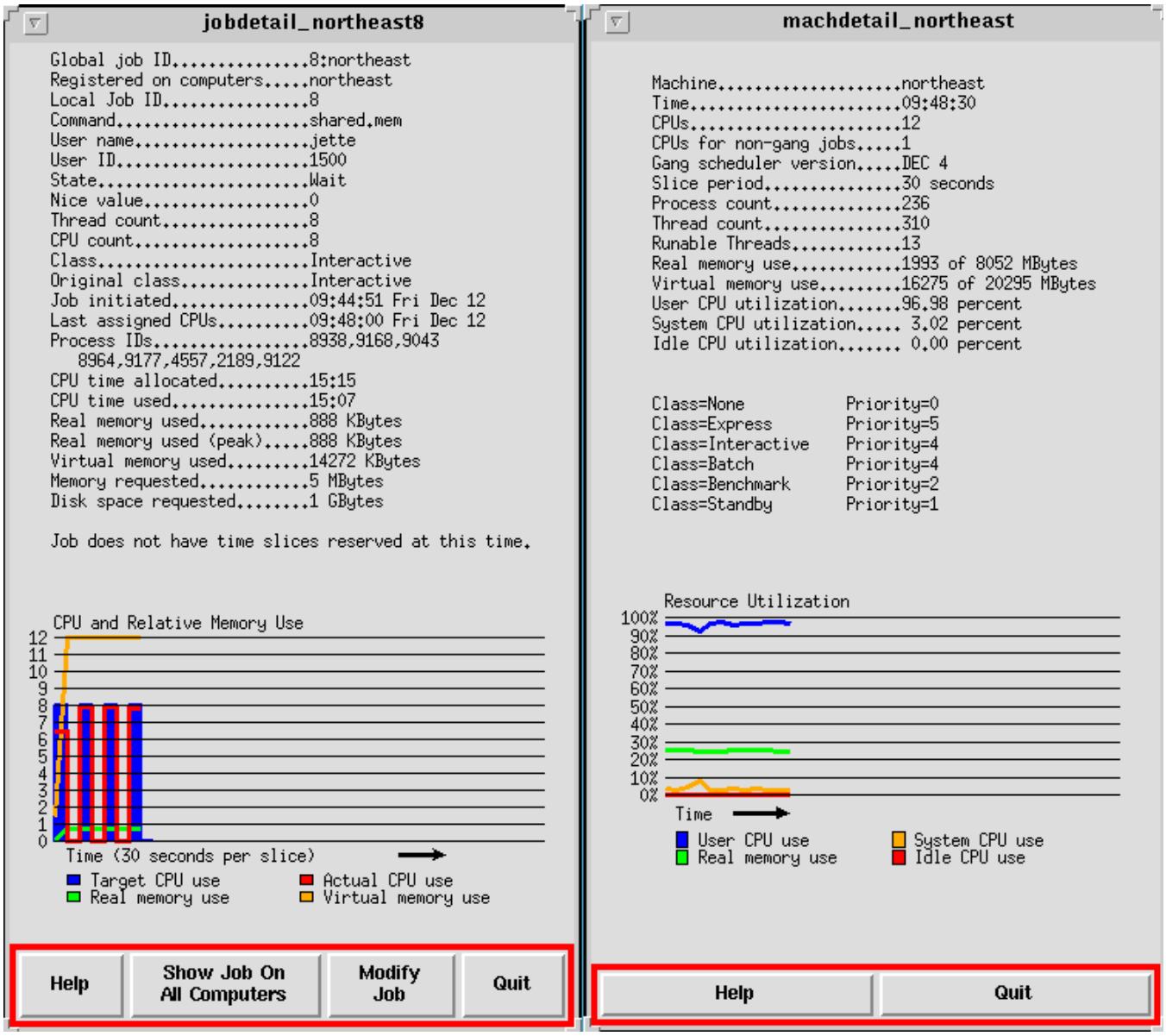


Figure 3: Gangster display of Digital program and machine status

The class scheduler provides a reasonable infrastructure for gang scheduling, but some performance enhancing tactics are used. A class is created by the gang scheduler for each registered parallel program on every computer the program will use. When processes are registered as a component of the program, their process IDs are added to the class. A class is allocated zero resources to stop the program, but this may not be completely effective if idle processors exist on the computer. In order to more effectively stop a program, the SIGUSR1 and SIGUSR2 signals are optionally used to pause and continue programs. The API permits an application to explicitly disable gang scheduler use of these signals if they are required by the program for other purposes, but doing so will reduce concurrency and may reduce its performance. These signals also permit a more tightly synchronized stopping of a program at the end of a time-slice than can be achieved by the class scheduler alone. Rather than waiting up to one second to propogate new scheduling information to the kernel, these signals can stop a program

immediately. Rather than allocating resources to a class in proportion to the number of processors desired, a higher target is specified and better overlap is achieved. This tactic effectively schedules auxiliary threads, which consume few compute cycles but are common on many applications. For example, a four thread program on an eight processor computer might be targeted to receive 60 percent of the resources rather than 50 percent. The actual percentage used varies system load and has been tuned to maximize parallel program overlap without causing significant reduction in responsiveness. The maximum resource allocation to all gang scheduled programs is limited to a configurable level. This may be used to insure that one or more processors are available to maintain overall system responsiveness.

Any gang scheduled program failing to utilize any CPU cycles for a configurable period of time, currently 10 minutes, will cease being gang scheduled and will revert to normal UNIX scheduling. Should the program resume consumption of CPU cycles, it will resume gang scheduling. This mechanism effectively addresses programs waiting for input, network traffic, or otherwise stopped. Any program faining to use any CPU cycles for an extended period, currently configured at 2 hours, is completely removed from the gang scheduler database.

One gang scheduler daemon executes on each computer. Programs spanning multiple computers contact the appropriate gang scheduler daemons to be preallocated specific time-slices on each computer. An Ousterhout [13] matrix is used to record these preallocated resources as shown in table 2. Each processor is represented by one column of the matrix and each row represents one time-slice. At prearranged times, the gang scheduler daemons allocate resources as specified in the Ousterhout matrix. The last row in the matrix, *time 4*, is followed by repeating the cycle from the top, *time 1*. In this gang scheduler implementation, the Ousterhout matrix describes a one hour schedule with the first time-slice occurring on the hour and subsequent time-slices at intervals configured when gang scheduler is built. All computers clocks must be synchronized to within a fraction of one second for concurrent scheduling to occur. LLNL uses a Network Time Protocol (NTP) for clock synchronization, although the Distributed Time Service (DTS) and other systems would equally satisfactory. The gang scheduler daemon uses an alarm to awake at the appropriate time and runs as user *root* to avoid being subject to class scheduling constraints.

	Computer East CPU 1	Computer East CPU 2	Computer West CPU 1	Computer West CPU 2
Time 1	Job A	Job A	Job B	Job B
Time 2	Job C	Job C	Job C	Job C
Time 3	Job A	Job A	Job B	Job B
Time 4	Job D	Job D	Job D	Job D

Table 2: Sample Ousterhout matrix

The gang scheduler is designed to provide each program with access to a similar quantity of processor cycles whether registered for gang scheduling or not. The number of time-slices, or entries in the Ousterhout matrix, allocated to a program spanning multiple computers is based upon the load on each computer at program initiation time. The program is allocated a percentage of Ousterhout matrix entries

equal to its proportion of threads on the most heavily loaded computer. For example, a program registering with the gang scheduler for four-way parallel on an eight processor computer with 12 other runnable threads should be allocated 25 percent of Ousterhout matrix entries on that computer, or four processors every other time-slice. A gang scheduler sub-system periodically may increase or decrease the number of time-slices pre-allocated to a program spanning multiple computers as system loads vary.

For programs which execute exclusively on one computer, scheduling decisions are made at the beginning of each time-slice. These programs lack entries in the Ousterhout matrix, but make use of available entries based upon current conditions. This permits the gang scheduler to rapidly respond to changes in the workload.

Time-slices are configured to be rather long, 30 seconds. While such a long time-slice reduce program responsiveness, it was necessitated by two factors. Class scheduler resource allocation targets require on the order of one second to be propogated to the kernel, resulting in unsatisfactory parallel program overlap for time-slice durations less than about 5 seconds. Second, many programs exceed one gigabyte in size and while context switching the processor may be performed in milliseconds, the time to refresh the cache may be on the order of hundreds of milliseconds and the time to context switch memory (paging one program from memory to disk and paging another program in the reverse direction) may be several seconds. In order to provide faster responsiveness, the execution of a newly initiated program may commence prior to the beginning of a new time-slice, if appropriate for the given workload. Also note that programs not registered for gang scheduling are not subject to these time-slices, but are scheduled using normal UNIX scheduling algorithms and compute resources not allocated to gang scheduled jobs.

Application Benefits

The most obvious benefit of gang scheduling to the application is the concurrent scheduling of required resources. Tightly synchronized threads of execution typically perform spin-wait at synchronization points rather than relinquishing their processors. Concurrent processor scheduling largely eliminates spin-wait time.

A second benefit of this gang scheduler implementation is that resources are allocated for much longer time periods than normally provided by Digital UNIX, permitting more efficient use of memory systems. Cache memory typically must be refreshed between context switches. By decreasing the frequency of context switches about 50 percent, the overhead of cache refreshing will be reduced. The applications typical of the LLNL workload utilize substantial memory resources. When several such applications are running concurrently, paging adversely impacts the performance of each.

While computer systems in which the number of executable threads never exceeds the number of processors can achieve similar performance for individual programs without the use of gang scheduling, this is difficult to achieve in practice. Computers designed as batch systems will have a regulated workload, but without some level of processor oversubscription, I/O bound programs will waste compute resources and even large compute-bound programs typically have I/O bound pre- and post-processing periods. A processor oversubscription rate of 50 percent (threads of queued work initiated on a computer equal to 150 percent of the processor count) largely eliminates idle processors for our workload. This will result in some competition for processors and even slight competition for processor resources can result in dramatic reduction in parallel program performance (five to 50 percent was not unusual for a range of benchmarks).

Application Consequences

While gang scheduling can provide the synchronization required by many applications, it can adversely impact performance of others. Reduced performance has been observed for both I/O bound programs and programs with severe memory contention. Most uniprocessor and SMP schedulers assign a high scheduling priority to processes waiting for I/O completion. This scheme maximizes the throughput of I/O bound programs without substantial impact upon processor availability. Since gang scheduling blocks the program's access to processors for some time-slices, the rate at which I/O requests can be issued and the overall program throughput is reduced. If the program is primarily compute bound, the Digital UNIX class scheduler will merely reallocate processors during periods of synchronous I/O and maintain high system utilization without substantial impact upon the individual program. Contention for the system's memory banks can also reduce a program's performance, particularly if its threads of execution are repeatedly writing to the same memory bank. This problem has been observed in only one parallel program performing repeated write instructions to a single memory location. The program was modified to eliminate the memory contention bottleneck and an overall improvement in throughput resulted. Since gang scheduling is provided only to programs explicitly registering for the service, gang scheduling may be easily avoided when appropriate. When in doubt, it is a simple matter of performing timing tests and comparing results to assess the benefit of gang scheduling.

Results

Performance characteristics of several benchmarks developed by Brooks and Warren [4] were utilized to assess the impact of gang scheduling. All benchmarks are tightly synchronized and compute bound, as is typical of the LLNL workload as a whole. The benchmarks were executed on a 12 processor Digital Alpha 8400 with 440 MHz clock and eight-way memory interleave. Twelve single-threaded application programs were running concurrently with these timing tests to simulate interference which might expected in a normal production environment. Table 3 and figure 4 show the performance of a 70 CPU second Gaussian elimination benchmark. Twenty executions were made at each thread count, alternating between gang and UNIX scheduling. Both mean and standard deviation values are report for MFLOP measurements based upon CPU time used. This benchmark experiences superlinear speedup due to the scaling of the cache size with processor count and high cache hit rates. Gang scheduling provided consistent program performance and scaling with increasing thread counts. Without gang scheduling, performance is good with small thread counts, but significant variation in performance occurred in each execution. Higher thread counts in some cases result in reduced program performance and the standard deviation in performance exceeds 10 percent in many cases. Gang scheduling benefits this benchmark partly through the synchronized processor allocation, but also through reduced the cache refresh overhead. The time period between synchronization points is inversely proportional to the thread count, at six threads the time is 1.10 seconds.

Thread Count	Gang Scheduled MFLOPS	Gang Scheduled Speedup	UNIX Scheduled MFLOPS	UNIX Scheduled Speedup
1	28.2 ± 0.4	1.00	29.9 ± 0.4	1.00
2	148.2 ± 2.1	5.25	127.4 ± 10.4	4.26
3	253.3 ± 3.4	8.98	163.6 ± 35.3	5.47
4	319.6 ± 2.3	11.33	287.4 ± 20.8	9.61
5	389.5 ± 7.5	13.81	280.1 ± 36.3	9.37
6	454.2 ± 8.2	16.11	268.7 ± 23.3	8.99
7	538.9 ± 13.7	19.11	226.0 ± 48.1	7.56
8	604.1 ± 9.8	21.42	104.5 ± 6.0	3.49
9	691.1 ± 10.4	24.51	136.6 ± 16.4	4.57
10	772.5 ± 9.0	27.39	145.3 ± 18.0	4.86
11	832.2 ± 10.5	29.51	191.9 ± 30.1	6.42

Table 3: Gaussian elimination benchmark performance

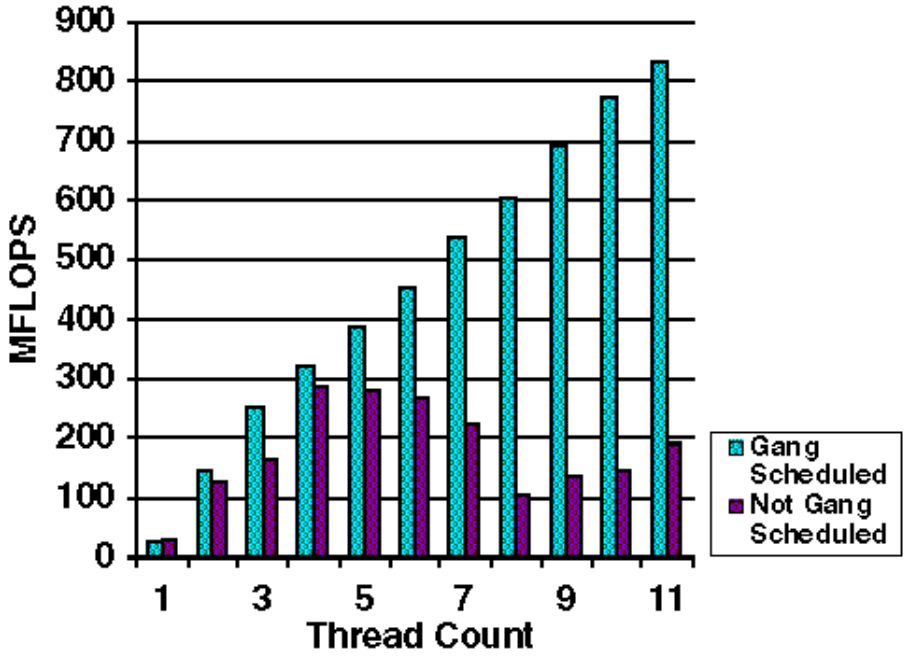


Figure 4: Gaussian elimination benchmark performance

The second benchmark investigated is a 15 CPU second matrix multiply benchmark. The memory requirements are sufficiently large to eliminate significant benefit of improved cache management. Benefit is provided primarily through synchronized processor assignment and reduced spin-wait time. The time period between synchronization points is inversely proportional to the thread count and is 2.67 seconds at six threads. Table 4 provides a summary of the results, but the results of individual timing tests are quite interesting. As expected, gang scheduling provided consistently good performance results. For most benchmark executions, normal UNIX scheduling provided similar performance to gang scheduling, but on occasion provided dramatically worse performance. This benchmark was executed twenty times at each thread count, alternating using UNIX and gang scheduling. An excerpt from the five thread benchmark log follows:

```
UNIX  615.16 MFLOPS
Gang  620.91 MFLOPS
UNIX  110.55 MFLOPS
Gang  623.83 MFLOPS
UNIX  110.64 MFLOPS
Gang  612.33 MFLOPS
UNIX  612.33 MFLOPS
```

Clearly the processes being managed by UNIX scheduling can result in substantial variation in spin-wait overhead. The timing tests at many thread counts demonstrated one or more abnormally low performance results.

Thread Count	Gang Scheduled MFLOPS	UNIX Scheduled MFLOPS
1	114.0 ± 0.4	111.8 ± 0.4
2	226.2 ± 0.9	227.5 ± 0.7
3	350.4 ± 0.2	347.8 ± 0.2
4	475.2 ± 2.7	469.7 ± 2.7
5	604.0 ± 4.8	403.1 ± 75.7
6	740.3 ± 5.7	737.5 ± 6.4
7	873.3 ± 8.7	728.6 ± 87.5
8	1002.4 ± 10.3	965.3 ± 23.7
9	1175.2 ± 12.8	1147.0 ± 23.5
10	1329.9 ± 16.8	1100.1 ± 124.9
11	1441.1 ± 14.8	1357.3 ± 72.5

Table 4: Matrix multiply benchmark performance

Most of our numerical analysis programs do benefit significantly from the cache and their performance characteristics seem to be best reflected by the Gaussian elimination benchmark. Typical parallel application programs executing in LLNL's normal production computing environment experience performance enhancements of five to 50 percent through gang scheduling and the customer response has been very positive. This gang scheduler has been in production use on some of LLNL's compute servers since July of 1997. No reduction in system utilization has been observed. During working hours, idle time is typically zero, system time only a few percent, and user time in in excess of 95 percent. System responsiveness is not noticeably reduced, although this has not been quantified.

In order to offload some work from MPP systems, a very popular Arbitrary Lagrange-Eulerian (ALE) hydrodynamics application was ported to the Digital cluster. Performance requirements dictated that this application be executed over sizable numbers of processors. Performance results for this application are shown in table 5 and figure 5 for both single computer and multiple computer executions. The Digital memory channel interconnect shows excellent performance here for cluster computing. Single computer and multiple computer executions show similar performance. The application displays near-linear speedup with gang scheduling even for large thread counts spanning 8 computers.

Thread Count	Run Time 1 Computer (Seconds)	Run Time 2 Computers (Seconds)	Run Time 4 Computers (Seconds)	Run Time 8 Computers (Seconds)
1	2313	NA	NA	NA
2	1235	1257	NA	NA
4	632	656	XX	NA
8	308	323	XX	XX
16	NA	XX	XX	XX
32	NA	NA	XX	XX

Table 5: ALE hydrodynamics application performance

Figure 5: ALE hydrodynamics application performance

Conclusion

Gang scheduling can provide substantially improved performance for tightly synchronized parallel programs in multiprogrammed environments, particularly those with large thread counts and substantial cache use. This can be accomplished without reduction in system utilization or noticeable reduction in responsiveness. Gang scheduling also provides the means of harnessing the power of an SMP cluster to address large-scale problems without sacrificing a multiprogramming capability. These results were achieved without binding threads to specific processors, although the benefit of this warrants further investigation.

Acknowledgments

Tony Verhulst of Digital Equipment Corporation developed the class scheduler infrastructure. Programs developed by Eugene Brooks, Mike Collette, Scott Futral, and Karen Warren were utilized to generate the performance results.

Appendix

The sample program shown below illustrates the program modifications required for gang scheduling. Equivalent FORTRAN subroutines are also available.

```
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include "GangUserAPI.h"
#define CPU_COUNT 2

main(int argc, char *argv[])
{
    int i;
    char host[MAXHOSTNAMELEN];

    gsRetVal rc;
    struct GangJobId my_job_id;
    struct GangResources gang_resources[1];
    struct GangResources *gang_resource_list[2];

    /* Clear job_id on first call, otherwise the calls will apply to an existing program */
    /* One can modify resource requirements of a program during its execution */
    bzero(&my_job_id, sizeof(my_job_id));

    /* Define resource requirements for each computer to be used */
    gethostname(host, sizeof(host));
    strcpy(gang_resources[0].machine, host);   /* Computer's name */
    gang_resources[0].cpu_count = CPU_COUNT;   /* CPU count desired */
    gang_resources[0].cpu_min   = CPU_COUNT;   /* Minimum CPU count acceptable */
    gang_resources[0].mega_mem  = 5;           /* Megabytes of memory desired (optional) */
    gang_resources[0].giga_disk = 1;           /* Gigabytes of disk desired (optional) */
    gang_resource_list[0]       = &gang_resources[0];

    gang_resource_list[1]       = NULL;         /* NULL terminated list of resources */

    /* Register the program */
    rc = GangJobRegister(&my_job_id, CLASS_INTERACTIVE, gang_resource_list);
    if (rc != gsSuccess) {
        printf("Error from GangJobRegister: %s\n", GangErrMsg(rc));
        exit(1);
    } /* if */
    printf("GangJobRegister completed successfully\n");   /* my_job_id is set */

    /* Fork processes as needed */
    for (i=1; i < CPU_COUNT; i++) {
        switch (fork()) {
        case -1:        /* Error */
            printf("Error forking process\n");
            exit(1);
        case 0:         /* Child */
            cpu_count = 0;
            break;
        default:        /* Parent */
            ;
        } /* switch */
    } /* for */

    /* Register each process */
    rc = GangProcAdd(&my_job_id, PROC_ID, getpid());
    if (rc != gsSuccess) {
```

```
        printf("Error from GangProcAdd: %s\n", GangErrMsg(rc));
        exit(1);
    } /* if */

    /* Run each process */
    printf("Running process %d \n", getpid());
    Parallel_Code();
    exit(0);
} /* main */
```

References

1. AT & T, **UNIX System V Release 4 Internals, Vol. 1**. AT&T, 1990, pp 2.4.1-2.4.21.

2. M. J. Bach, **The Design of the UNIX Operating System**. Prentice-Hall Inc., 1986, pp 247-268.

3. J. M. Barton and N. Bitar, **A Scalable Multi-discipline, Multiple-processor Scheduling Framework for IRIX**. Job Scheduling Strategies for Parallel Processing, Edited by: Feitelson, D.G.; Rudolph, L. Berlin, Springer Verlag, 1995. Lecture Notes in Computer Science, Vol 949, pp 45-69.

4. E. D. Brooks III and K. H. Warren, **A Study of Performance on SMP and Distributed Memory Architectures Using A Shared Memory Programming Model**. Proceedings of SuperComputing97, Nov 1997.

5. D. G. Feitelson, **A Survey of Scheduling in Multiprogrammed Parallel Systems.** Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

6. D. G. Feitelson and M. A. Jette, **Improved Utilization and Responsiveness with Gang Scheduling**. IPPS ’97 Workshop on Job Scheduling Strategies for Parallel Processing, Apr 1997, pp 238-261.

7. B. Gorda and R. Wolski, **Timesharing massively parallel machines**. International Conference on Parallel Processing, volume II, Aug 1995, pp 214-217.

8. B. C. Gorda and E. D. Brooks III, **Gang Scheduling a Parallel Machine**. Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.

9. A. Gupta, A. Tucker, and S. Urushibara, **The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications** . Proceedings of ACM SIGMETRICS, pp 120-132, May 1991.

10. M. Jette, D. Storch and E. Yim, **The Gang Scheduler - Timesharing the Cray T3D**. Cray User Group, Mar 1996, pp 247-252.

11. M. Jette, **Performance Characteristics of Gang Scheduling in Multiprogrammed Environments**. Proceedings of SuperComputing97, Nov 1997.

12. R. N. Lagerstrom and S. K. Gipp, **PScheD Political Scheduling on the CRAY T3E**. Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing Job Scheduling Strategies for Parallel Processing, Edited by: Feitelson, D.G.; Rudolph, L. Berlin, Springer Verlag, 1997.

Lecture Notes in Computer Science, Vol 1291, pp 117-138.

13. J. Ousterhout, **Scheduling techniques for concurrent systems**. Proceedings of the Third International Conference on Distributed Computer Systems, Oct 1982, pp 22-30.

14. M. K. Seager and J. M. Stichnoth, **Simulating the Scheduling of Parallel Supercomputer Applications**. Technical Report UCRL-102058, Lawrence Livermore National Laboratory, Sep 1989.

Work performed under the auspices of the U.S. DOE by LLNL under contract W-7405-ENG-48.

Document Number UCRL-MI-????.

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551