

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/3300984>

Adaptive parallel job scheduling with flexible coscheduling

Article in IEEE Transactions on Parallel and Distributed Systems · December 2005

DOI: 10.1109/TPDS.2005.130 · Source: IEEE Xplore

CITATIONS

56

READS

107

4 authors, including:



Eitan Frachtenberg

88 PUBLICATIONS 2,479 CITATIONS

[SEE PROFILE](#)



Fabrizio Petrini

IBM

155 PUBLICATIONS 4,891 CITATIONS

[SEE PROFILE](#)



Juan Esteban Fernández

Complutense University of Madrid

23 PUBLICATIONS 486 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Random Access Benchmark [View project](#)



Network Interface Cards (NICs) as First-Class Citizens [View project](#)

Adaptive Parallel Job Scheduling with Flexible CoScheduling*

Eitan Frachtenberg¹, Dror G. Feitelson², Fabrizio Petrini¹ and Juan Fernandez¹

¹ CCS-3 Modeling, Algorithms, and Informatics Group	² School of Computer Science and Engineering
Computer and Computational Sciences (CCS) Division	The Hebrew University, Jerusalem, Israel
Los Alamos National Laboratory (LANL)	feit@cs.huji.ac.il
{eitanf,fabrizio,juanf}@lanl.gov	

Abstract

Many scientific and high-performance computing applications consist of multiple processes running on different processors that communicate frequently. These applications can suffer severe performance penalties if their processes are not all coscheduled to run together, due to their synchronization needs. Two common approaches to coschedule jobs are batch scheduling, wherein nodes are dedicated for the duration of the run, and gang scheduling, wherein time slicing is coordinated across processors. Both work well when jobs are load-balanced and make use of the entire parallel machine. However, this is rarely the case, and most realistic workloads consequently suffer from both internal and external fragmentation, where resources and processors are left idle because jobs cannot be packed with perfect efficiency. This leads to reduced utilization and sub-optimal performance. Flexible CoScheduling (FCS) addresses this problem by monitoring each job's computation granularity and communication pattern, and scheduling them based on their synchronization and load-balancing requirements. In particular, jobs that do not require stringent synchronization are identified, and are not coscheduled; instead, these processes are used to reduce fragmentation. FCS has been fully implemented on top of the

*This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36.

STORM resource manager on a 256-processor Alpha cluster, and compared to batch, gang, and implicit coscheduling algorithms. This paper describes in detail the implementation of FCS and its performance evaluation with a variety of workloads, including large-scale benchmarks, scientific applications, and dynamic workloads. The experimental results show that FCS saturates at higher loads than other algorithms (up to 54% higher in some cases), and displays lower response times and slowdown than the other algorithms in nearly all scenarios.

Keywords: *Cluster computing, load balancing, job scheduling, gang scheduling, parallel architectures, Flexible coscheduling*

1. Introduction

Clusters of workstations are steadily growing larger and more prevalent. Although cluster hardware is improving in terms of price and performance, cluster utilization remains poor. Load imbalance is arguably one of the main factors that limits resource utilization, in particular in large-scale clusters [5]. Load imbalance can have a marked detrimental effect on many parallel programs. A large subset of high performance computing (HPC) software can be modeled using the bulk-synchronous parallel (BSP) model. In this model a computation involves a number of *supersteps*, each having several parallel computational threads that synchronize at the end of the superstep [6, 13, 23]. A load imbalance can harm the performance of the whole parallel application because each thread of computation requires a different amount of time to complete, and the entire program must wait for the slowest thread before it can synchronize. Since these computation/synchronization cycles are potentially executed many times throughout the lifetime of the program, the cumulative effect on the application run time and the system resource utilization can be quite high [20].

Load imbalance has three main sources: application imbalance, workload imbalance, and heterogeneity of hardware resources. Application load imbalance occurs when different parallel threads of computation take varying times to complete the superstep. This can occur either as a result of poor programming, or more typically, by a data set that creates uneven loads on the different threads.

Even when using well-balanced software load imbalances can occur. This can happen, for instance, when the compute nodes are not entirely dedicated to the parallel computation because they are also being used for local user- or system-level programs, or because the resource management system cannot allocate an even workload to all processors. This uneven taxing of resources creates a situation where some parts of the parallel program run slower than others, and a load imbalance occurs [20].

Load imbalance can also be generated by heterogeneous architectures in which different nodes have different computational capabilities, different memory hierarchy properties, or even a different number of processors per node. Two examples of such situations are grid computing and HPC systems that accrue additional processing nodes over a period of time, thus taking advantage of technological improvements.

The traditional approach to tackle load imbalance is at the application level: the programmer tries to balance the resources by changing the structure of the parallel program. This approach is usually time consuming and yields diminishing returns after an initial phase of code restructuring and optimizations. In fact, there are some problems that are inherently load-imbalanced. This approach is also not economically feasible with legacy codes. For example, the Accelerated Strategic Computing Initiative (ASCI) program [25] invested more than a billion dollars in the last few years in parallel software, with a yearly increase of several hundreds of millions of dollars.

An alternative approach is to attack load imbalance at the runtime level. Rather than optimizing a single parallel job, we can coschedule (time-slice on the same set of processors) multiple parallel jobs and try to compensate for the load imbalance within these jobs. This approach is also better suited to handle complex workloads and/or heterogeneous architectures. Ideally, we would like to transform a set of ill-behaved user applications into a single load-balanced system-level workload. This approach has the appealing advantage that it does not require any changes to existing parallel software, and is therefore able to deal with existing legacy codes. For example, coscheduling algorithms such as Implicit Coscheduling (ICS) [3], Dynamic Coscheduling (DCS) [21] or Coordinated Coscheduling (CC) [4] can potentially alleviate load imbalance and increase resource utilization.¹ However, they are not always able to handle all job types because they do not rely on global coordination. On the other hand, global

¹These algorithms are distinguished by where the coordination is inferred: at sender side, receiver, or both, respectively.

resource coordination and job preemption can have a significant cost, if implemented using software mechanisms only [12, 17].

In this paper we show that it is possible to substantially increase the resource utilization in a cluster of workstations and to effectively perform system-level load balancing. We introduce an innovative methodology that can dynamically detect and compensate for load imbalance, called Flexible CoScheduling (FCS). Dynamic detection of load imbalances is performed by (1) monitoring the communication behavior of applications, (2) defining metrics for their communication performance that attempt to identify possible load imbalances, and (3) classifying of the applications according to these metrics. On top of this, we propose a coscheduling mechanism that uses this application classification to execute scheduling decisions. The scheduler strives to coschedule those processes that would most benefit from it, while scheduling other processes to increase overall system utilization and throughput. This approach does not alleviate the specific situation of an application that suffers from load imbalances. Obviously, any given application receives the best service when running by itself on a dedicated set of nodes. However, the proposed approach will prevent each job from wasting too many system resources, and the overall system efficiency and responsiveness will be improved, which in turn lowers the single application's waiting time.

We demonstrate this methodology with a streamlined implementation on top of STORM (Scalable TOol for Resource Management) [9]. The key innovation behind STORM is a software architecture that enables resource management to exploit low-level network features. As a consequence of this design, STORM can enact scheduling decisions, such as a global context switch or a heartbeat, in a few hundreds of microseconds across thousands of nodes. Thus, STORM avoids much of the non-scalable overhead associated with software-only versions of gang scheduling. An important innovation in FCS is the combination of a set of local policies with the global coordination mechanisms provided by STORM, in order to coschedule processes that have a high degree of coupling.

In preliminary work we presented initial benchmark results for FCS, running on a cluster of Pentium-III machines [8]. This paper extends that work with new experiments on a larger experimental platform and with new metrics, and using more realistic applications and workloads. Additionally, FCS was

further tuned and simplified, and provides better performance results. In the experimental section, we provide an empirical evaluation which ranges from simple workloads that provide insights on several job scheduling algorithms to experiments with real applications representative of the ASCI workload.

2. Flexible Coscheduling

To address the problems described above we propose a novel scheduling mechanism called Flexible CoScheduling (FCS). The main motivation behind FCS is the improvement of overall system performance in the presence of load imbalance, by using dynamic measurement of applications' communication patterns and classification of applications into distinct types. Some applications strictly adhere to the BSP model with balanced, fine-grained communications. Others deviate from this model, either due to little communication or to inherent load imbalances. We can therefore restate FCS' goal as identifying the proper synchronization needs of each application and process, and trying to optimize the entire system's performance while addressing these needs. FCS is implemented on top of STORM [9], which allows for both global synchronization through scalable global context switch messages (heartbeats) and local scheduling by a daemon run on every node, based on its locally-collected information. User-level scheduling (as opposed to kernel-level) incurs some additional overhead, but eliminates the need to communicate frequent scheduling information to the kernel [2].

2.1. Process Classification

FCS employs dynamic process classification and schedules processes using this class information. Processes are categorized into one of three classes (Figure 1):

1. *CS* (coscheduling): These processes communicate often, and must be coscheduled (gang-scheduled) across the machine to run effectively, due to their demanding synchronization requirements.
2. *F* (frustrated): These processes have enough synchronization requirements to be coscheduled, but due to load imbalance, they often cannot make full use of their allotted CPU time. This load imbalance can result from any of the reasons detailed in the introduction.

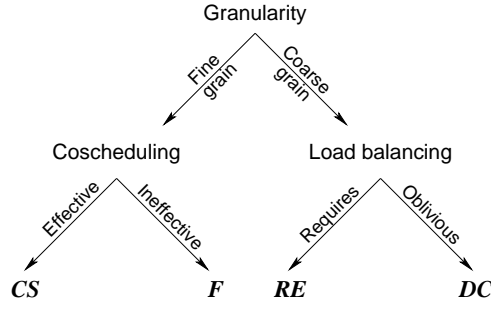


Figure 1. Decision tree for process classification

3. *DC* (don't-care): These processes rarely synchronize, and can be scheduled independently of each other without penalizing the system's utilization or the job's performance. For example, a job using a coarse-grained workpile model would be categorized as *DC*. We include in *DC* also the processes of a fourth class, *RE* (rate-equivalent). This class is characterized by jobs that have little synchronization, but require a similar (balanced) amount of CPU time for all their processes. Since detection of *RE* processes cannot be made in run-time with local information only, they are classified as *DC* instead, due to their low synchronization needs.

Figure 1 shows the decision tree for process classification. Each process is evaluated at the end of its timeslot.² If a process communicates at relatively coarse granularity, it is either a *DC* or *RE* process and classified as *DC*. Otherwise, the process is classified according to how effectively it communicates when coscheduled: if effective, it is a *CS* process. Otherwise, some load imbalance prevents the process from communicating effectively, and it is considered *F*. To estimate the granularity and effectiveness of a process' communication, we modified the MPI library so that blocking communication calls take time measurements and store them in a shared-memory area, where the scheduling layer can access them. Only synchronous (blocking) communication calls are monitored, since non-blocking communications do not require tight synchronization and need not affect scheduling (thus a call to `MPI_Isend()` is non-blocking, but `MPI_Wait()` is considered blocking).

²In strict gang scheduling each job is assigned a dedicated timeslot, and can only be run in that slot. FCS also assigns a timeslot to each job, but local scheduling decisions can cause the job to run in other timeslots as well, possibly sharing them with other jobs. We call the original timeslot to which a process is mapped the "assigned timeslot".

Processes of the same job will not always belong to the same class. For example, load imbalance or system heterogeneity can lead to situations in which one process needs to wait more than another. To allow for these cases and to avoid global exchange of information, processes are categorized on an individual basis, rather than per-job.

This classification differs in two important ways from a similar one suggested by Lee et al. [14]. First, we differentiate between the *CS* and *F* classes, so that even processes that require gang scheduling would not tax the system too much if heterogeneity prevents them from fully exploiting coscheduling. Second, there is no separate class for *RE* applications. These are indistinguishable (from the scheduler's point of view) from *DC* processes, and are scheduled in the same manner. The classification also differs from the one suggested by Wiseman [24], which is based on CPU utilization and is done at the job rather than the process level.

2.2. Scheduling

The scheduling principles in FCS are as follows:

- *CS* processes are always coscheduled and should not be preempted.
- *F* processes need coscheduling but are preempted if synchronization is ineffective.
- *DC* processes impose no restrictions on scheduling.

The infrastructure used to implement this scheduling algorithm (STORM) is based on an implementation of conventional gang scheduling [9]. A single system-wide manager, the machine manager daemon (MM), packs the jobs into an Ousterhout matrix. It periodically sends multi-context-switch messages to the node manager daemons (NM), instructing them to switch from one timeslot to another. A crucial characteristic is that the node managers are not obligated to comply: they are free to make their own scheduling decisions based on their local measurements and classifications.

Algorithm 1 shows the behavior of the node manager upon receipt of a multi-context-switch message. The basic idea is to allow the local operating system the freedom to schedule *DC* processes according to its usual criteria (fairness, I/O considerations, etc.), as well as to use *DC* processes to fill in the gaps

Algorithm 1: Context switch algorithm for FCS

```
// context_switch: switch from one process to another process
// Invoked for each processor by a global multi-context-switch
procedure context_switch (current_process, next_process)
begin
  if current_process == next_process then return
  if type of next_process is CS then
    suspend whatever is running on this PE
    run next_process for its entire time slot
    use polling for synchronous communications
  else
    resume DC and F processes belonging to this PE
    let local OS scheduler schedule all processes
    use spin-blocking in synchronous communications
    if next_process is of type F
      prioritize it over all other processes.
  end
end
end
```

that *F* processes create because of their synchronization problems. An *F* process that waits for pending communication does not block immediately, but rather spins for some time to avoid unnecessary context switch penalties, as in ICS [3].

This scheduling algorithm represents a new approach to dynamic coscheduling methods, since it can benefit both from scalable global scheduling decisions and local decision based on detailed process statistics. Furthermore, it differs from previous dynamic coscheduling methods like DCS [21] and ICS in that:

1. A *CS* process in FCS cannot be preempted before the timeslot expires even if a message arrives for another process (processes classified as *CS* have shown that it is not worthwhile to deschedule them in their timeslot, due to their fine-grained synchronization). Blocking events therefore do not cause yielding of the CPU.
2. The local scheduler's decision in choosing among processes in the *DC* timeslots and *F* gaps is affected by the communication characterization of processes, which could lead to less-blocking processes and higher utilization of resources. Another improvement over the work presented in [8]

	Name	Description
MPI monitoring	T_{cpu}	CPU time since last reset (sec)
	T_{comm}	Total time waiting for blocking commun. to complete since last reset (sec)
	C_{comm}	Count of blocking commun. operations since last reset
	\overline{T}_{cpu}	Average CPU time per commun.: $\frac{T_{cpu}}{C_{comm}}$
	\overline{T}_{comm}	Average wait per commun.: $\frac{T_{comm}}{C_{comm}}$
Scheduling	$class$	Either CS , F , or DC
	$cslots$	Assigned timeslots in current class
	$tslots$	Total assigned timeslots since start
	g	Granularity (sec): $\overline{T}_{cpu} + \overline{T}_{comm}$

Table 1. FCS parameters

is that CS processes are no longer allowed to run in the F slots. Empirical evaluation has shown that this sharing typically results in worse overall performance. This is probably due to the fact that both competing processes are essentially fine-grained, and cannot run both well at the same time, while on the other hand context-switch and cache flushing issues degrade their performance.

2.3. FCS Parameters

There are three types of parameters used in FCS:

1. Process characteristics measured by the MPI layer, summarized in Table 1. The “reset” mentioned in the table is either a class change or a predetermined age expiration.
2. Parameters measured or determined by the scheduling layer, also detailed in Table 1.
3. Algorithm and run-time constants, shown in Table 2.

Measurements are taken whenever a process is running. For highly synchronized processes, we have verified that they typically make progress only in their assigned slots, so the measurements indeed reflect their behavior when coscheduled. For other processes the assigned slot does not have a large effect on

Name	Description	Value
T_{slice}	Timeslice quantum	100 ms
T_{spin}	Spin time for spin-block communications	120 μs
$cslots_{MIN}$	Minimum value of $cslots$ for process to be evaluated for a class change	20
DC_{thresh}	DC granularity threshold: above this value process is DC	1s
CS_{thresh}	CS granularity threshold: below this value process is CS	2 ms
F_{thresh}	threshold of computation granularity to identify processes waiting for communication as F	$0.85 \times CS_{thresh}$
$tslots_{MAX}$	Maximum value of $tslots$, after which a reset to class CS is forced	32768

Table 2. FCS constants and values used in experiments.

their progress, except possibly for F processes that get a higher priority in their slot. For DC and F processes, the assigned timeslot is used mainly to track the age of a process using the $cslots$ and $tslots$ counters.

The following are some of the considerations that led us to choose the values in Table 2:

- T_{slice} was chosen to be low enough to enable interactive responsiveness, and high enough to have no noticeable overhead on the applications, as measured in [7].
- T_{spin} was chosen to be high enough to accommodate twice the average communication operation (In our setup, $\approx 60 \mu s$ [19]), and low enough so that resources are not wasted unnecessarily.
- $cslots_{MIN}$ should allow enough time for some initializations to occur, but without overly delaying proper classification.
- CS_{thresh} : It was found that proper classification has the most effect for processes with a granularity finer than $\approx 5 ms$ on this architecture.

All constants, and in particular the last two, were found by careful tuning and testing on all our hardware and software combinations, to offer good average performance across the board [7, 9].

2.4. Characterization Heuristic

Algorithm 2 shows how a process is reclassified. This algorithm is invoked for every process that has just finished running in its assigned timeslot, so this happens at deterministic, predictable times

Algorithm 2: Classification function for FCS

```
// re-evaluate, and possibly re-classify the process
// using FCS parameters and measurements
procedure FCS_reclassify
begin
  old_class = class
  if cslots < cslots_MIN
    return // Not running long enough in current class
  if tslots mod tslots_MAX == 0 OR g < CS_thresh
    class = CS // Reset or change class back to CS
  else if g < DC_thresh AND  $\overline{T_{cpu}} < F_{thresh}$ 
    class = F // Communication too slow
  else class = DC // Coarse granularity

  if class != old_class
    cslots = 0
end
```

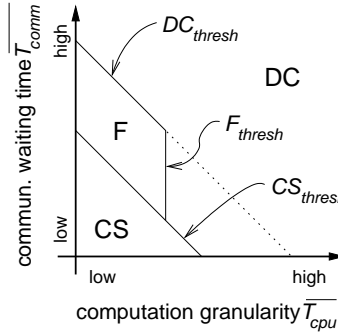


Figure 2. Phase diagram of classification algorithm

throughout the machine. Thus, when the time to reset a process to class *CS* arrives, it is guaranteed that all the processes of the same job will be reset together (otherwise they might not actually be coscheduled).

The algorithm can be illustrated with the phase diagram shown in Figure 2. Recall that the granularity g is defined as the average time per iteration, which is the sum of the average computation and communication times. Therefore constant granularity is represented by diagonals from upper left to lower right. *CS* processes occupy the corner near the origin, whereas *DC* processes are those that are far from this corner. *F* processes are those that should be in the corner because of their low $\overline{T_{cpu}}$, but suffer from a relatively high $\overline{T_{comm}}$.

2.5. Implementation Framework

We have implemented FCS and several other scheduling algorithms in STORM [9], a scalable, flexible resource management system for clusters, implemented on top of various Intel- and Alpha-based architectures. STORM exploits low-level collective communication mechanisms to offer high-performance job launching and management. As mentioned above, the basic software architecture is a set of daemons, one for the entire machine (machine manager, or MM), and an additional one for each node (node manager, or NM). This architecture allows the implementation of many scheduling algorithms by “plugging-in” appropriate modules in the MM and NM. Thus, FCS was added to STORM with two relatively simple enhancements: an MPI hook to measure and export information on processes’ synchronous communication to the NM, and a module in the NM that translates this information into a classification, and schedule processes based on their class. Note that the MM was not modified relative to GS.

2.6. Implementation Issues

Measuring process statistics can be both intrusive and imprecise if not performed carefully. It is important to take measurements with as little overhead as possible, without significantly affecting or modifying the code. To realize this goal, we implemented a lightweight monitoring layer that is integrated with MPI. Synchronous communication primitives in MPI call one of four low-latency functions to note when the process starts/ends a synchronous operation and when it enters and exits blocking mode. Applications only need to be re-linked with the modified MPI library, without any change. The accuracy of this monitoring layer has been verified using synthetic applications for which the measured parameters are known in advance, and found to be precise within 0.1%.

The monitoring layer updates the MPI-level variables shown in Table 1. These variables reside in shared memory, to allow the NM to read them without issuing a system call. While this transfer is asynchronous and a lag could exist between the actual communication event and the time the NM gathers the information, these parameters converge quickly to an accurate picture of the process’ characteristics.

To count communication events (C_{comm}), we employed the following guideline: multiple communi-

cation events with no intervening computation are considered to be a single communication event. This heuristic works very accurately as long as the granularity of the process is greater than that of the local operating system — otherwise the computation intervals are too short to be registered by the operating system. We found this value to typically be around twice the Linux 1/HZ value (the timer interrupt frequency), which defaults to $\approx 1\text{ ms}$ on our cluster. To detect finer granularities, we increased the HZ value to 4096.

3. Synthetic Scenarios

In this section we analyze the characteristics of FCS under four basic synthetic benchmarks, and compare it to three scheduling algorithms: batch scheduling (first-come-first-serve, or FCFS), gang scheduling (GS), and spin-block (SB). SB is an instance of implicit coscheduling (ICS) [3], and has been shown to perform on a par with other implicit algorithms without global coordination, as in gang scheduling [1, 4]. With SB, processes that wait for a synchronous communication poll for a given interval, and only if the communication has not completed by this time they block (in contrast, gang scheduled processes always busy-wait). Thus, processes tend to self-synchronize across the job, so relatively good coordination is achieved without the need for an explicit coscheduling. In ICS the spin time can be adaptive [3], thus decreasing inefficiencies resulting from spinning too long. In our implementation of SB we chose a small constant time for spinning ($120\text{ }\mu\text{s}$), so that very little time is wasted. Note that typical communication operations with this hardware and software setup complete in far less than this time (a few tens of μs), so if two communicating processes are coscheduled, they are almost guaranteed to complete the communication within this time interval.

We use two metrics to compare the performance of different scheduling algorithms [6]:

- Turnaround time - the total running time (in seconds) of the entire workload.
- Average response time - the mean time it takes a job to complete running from the time of submittal (enqueueing), which is not necessarily the actual execution time.

Turnaround time is considered a system-centric metric, since it describes the reciprocal of the system's

throughput. Response time on the other hand is more of interest to users, that would like to minimize the time they wait for their individual jobs to complete. In practice, it is difficult to discuss these metrics in isolation, since with real dynamic workloads, various factors and feedback effects create interactions between the metrics [6, 7]. However, the four scenarios we describe in this section are simple enough to allow a comprehensive understanding of the factors involved. We believe that this set of synthetic tests covers a wide spectrum of basic workload combinations.

For all four scenarios, we use a simple synthetic application as the “building-block” of the workload. This job is modeled after the BSP model [23], containing a simple loop with some computation followed by a communication phase in a nearest neighbor, ring pattern. The computation granularity (per iteration) in the basic case was chosen to be 1 *ms*, which is considered to be a relatively fine granularity when compared to real scientific applications’ performance in the given hardware environment [10]. When running in isolation, the basic “building block” job takes approximately 60 *s* to complete.

The hardware used for the experimental evaluation was the “Wolverine” cluster at LANL. This cluster consists of 64 HP Alpha compute nodes and a dual-rail QsNet network [19]. Each compute node contains four 833 MHz EV6 processors, 8 GB of ECC RAM, two independent 32 MHz/64-bit PCI buses, two Quadrics QM-400 Elan3 NIC [19] for the data network, and a 100 Mbit Ethernet network adapter for the management network. All the nodes run Red Hat Linux 7.3 with Quadrics kernel modifications and user-level libraries. Our tests used a simple workload of 2-3 jobs arriving concurrently using half the cluster, 128 processors on 32 nodes. This cluster architecture and size combined with the very fine granularity is a good stress test for scheduling algorithms: This configuration is not conducive for good performance with fine-grained applications and/or global context switching, because of the machine’s high sensitivity to noise created by system daemons and our own scheduling system [20]. To mitigate this problem we used a context-switch interval of 100 *ms*, which is high enough to reduce most of the overhead and noise effects, while still being responsive enough to be considered interactive. Every experiment was repeated five times, with the best result taken for each algorithm.

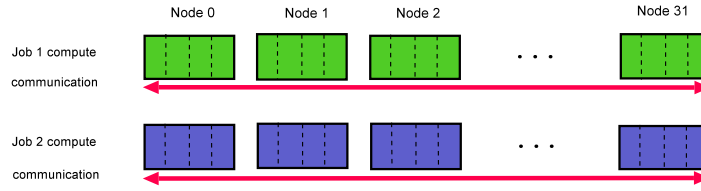


Figure 3. One iteration of two “building-block” jobs

3.1 Balanced jobs

Many HPC applications are latency bound, in the sense that they synchronize often with short messages. These synchronous applications require that all their processes be coscheduled to communicate effectively: if another application or system daemon interrupts their synchronization, large skews can develop that significantly hamper their performance [20].

In the first scenario, we emulate such situations by running two identical, fine-grained jobs concurrently. Figure 3 depicts the run time *per iteration*, which is balanced and equal for both jobs.³ The following table presents the results for running this workload, giving the termination time in seconds for each job and for the complete set. It also shows the total turnaround and mean response times, all in seconds.

Algorithm	Job 1	Job 2	Turnaround	Mean response
FCFS	60	120	120	90
GS	124	124	124	124
SB	126	134	134	130
FCS	125	126	126	126

Since synchronous balanced jobs require a dedicated environment to run effectively, FCFS scheduling and GS offer the best performance. SB scheduling shows noticeable slowdown when compared to the others, due to the lack of global coordination. FCS exhibits performance comparable to that of GS, since all processes are classified as CS, and are therefore gang-scheduled. Still, total turnaround time is slightly

³In this and the following figures, the compute and communicate phases are shown per iteration. Scheduling determines which processes run together and for how many iterations.

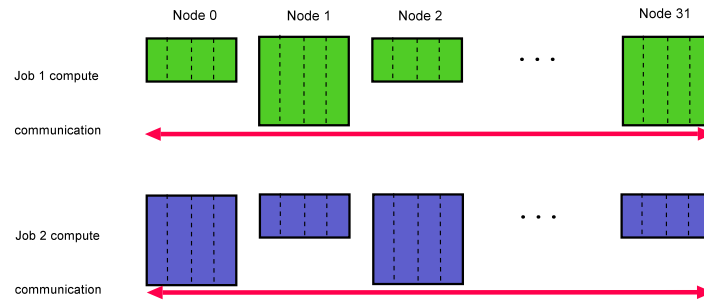


Figure 4. Two load-imbalanced jobs

higher than that of GS, due to the added overhead of process classification. When considering response time, batch scheduling is the only algorithm that has a significant advantage over the other algorithms, since job 1 that runs in isolation terminates quickly and lowers the average.

3.2 Load-imbalanced jobs

This scenario represents a simple load imbalanced case with two complementing jobs, as depicted in Figure 4. Every alternating node (4 processes) in the first job computes twice as much *per iteration* as the other node, while the situation is reversed for job 2⁴. The faster processes compute for the same amount of time as in the previous scenario. In a sense, this workload represents the exact opposite of the previous one, where jobs need a dedicated partition to communicate effectively. In contrast, these unbalanced jobs are guaranteed to waste compute resources when running in isolation. The following table shows the performance of each scheduling algorithm.

Algorithm	Job 1	Job 2	Turnaround	Mean response
FCFS	120	240	240	180
GS	244	245	245	245
SB	193	194	194	194
FCS	197	197	197	197

⁴In reality, the speed ratio is slightly over 2:1, to bring the total runtime of each job to 120 s. The gap is produced by the communication time, which is unchanged, requiring additional computation to increase the run time.

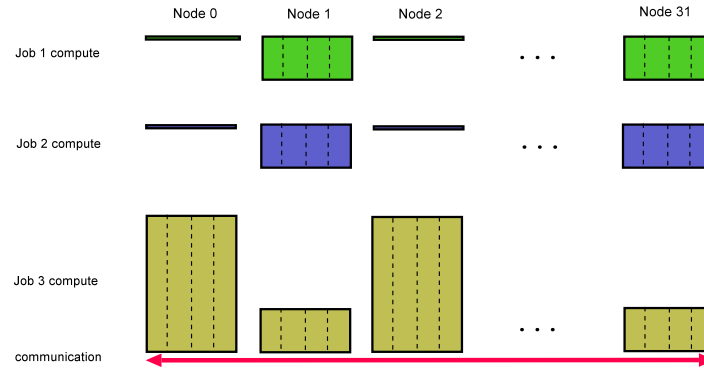


Figure 5. Complementing jobs

It can be seen from the data that both FCFS and GS take almost twice as much time to run each job (compared to the previous scenario), whereas the total amount of computation per job is only increased by 50%. SB does a much better job at load-balancing, since the short polling interval allows the algorithm to yield the CPU when processes are not coscheduled, giving the other job a chance to complete its communication and wasting little CPU time. FCS is also successful in exploiting these computational holes. After a brief interval, it classifies the first job's processes in alternating nodes as $F, F, F, DC, DC, DC, DC, \dots$, and the second job's as $DC, DC, DC, DC, F, F, F, \dots$. The resulting scheduling is effectively the same as SB's, with the exception that F processes are prioritized when their assigned slot is the active one. The total turnaround time is similar to SB's, and represents close to optimum resource utilization: both jobs complete after running for $\approx 159\%$ of the time it took the previous scenario, which corresponds to $\approx 6\%$ more than the total amount of compute work.

The response time metric again shows some preference to batch scheduling, although FCS and SB are not far behind, due to their lower turnaround time. GS exhibits the approximately same turnaround time as FCFS, but since all the process terminate concurrently, the mean response time is actually higher.

3.3 Complementing jobs

The third scenario exposes the ability of various algorithms to pack jobs efficiently in a more imbalanced workload. It consists of two non-communicating 60 s jobs and one communicating 180 s job that

are arranged so that alternating nodes run for different amounts of time, taking a total of 180 s per processor when aggregated over all processors (see Figure 5). An optimal scheduler should pack all these jobs so that the total turnaround time does not exceed that of the third job when run in isolation (assuming zero context-switch overhead). The following table again shows the performance for each algorithm.

Algorithm	Job 1	Job 2	Job 3	Turnaround	Mean response
FCFS	60	121	301	301	161
GS	185	186	308	308	226
SB	144	142	244	244	177
FCS	192	193	197	197	194

Once more, FCFS and GS exhibit similar turnaround time — the combined run time of all the jobs run in isolation. SB shows some ability to load-balance the jobs, reducing the overall turnaround and response times, but since it lacks a detailed knowledge of the processes’ requirements, it can only go so far — Job 3 still shows a significant slowdown ($\approx 36\%$) when compared to FCFS. With FCS the situation is even better. The scheduler classifies all the processes as *DC*, except for the faster processes of job 3 which are classified as *F*. As such, they receive priority in their timeslot, and thus the total runtime of job 3 suffers a slowdown of $\approx 9\%$ from the interference of the other jobs, which pack neatly into the other timeslices. This reduces not only the turnaround of this workload to $\approx 9\%$ of the optimal packing value (paying some price to overhead), but also minimizes the mean response time, which is competitive with that of FCFS. A reordering of the jobs would not affect FCS’ metrics, while it would worsen FCFS’ response times. Time sharing algorithms are not as sensitive to job order, which becomes an advantage when the order is not known in advance.

3.4 Balanced and Imbalanced Job Combination

The last synthetic scenario is designed to expose the interaction between synchronous balanced and imbalanced jobs in a mixed workload. This situation might occur whenever a machine is running more than one type of application, or with different data sets that have different load balancing properties.

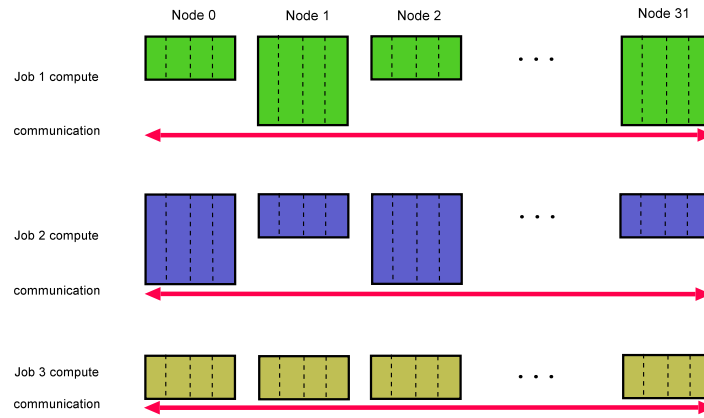


Figure 6. Balanced and imbalanced job combination

Even if the workload of a parallel machine is composed of only balanced applications, this situation can occur whenever job arrivals and sizes are dynamic. For example, in a time sharing system, different nodes might run different numbers of jobs, thus creating a dynamic imbalance.

We encapsulate some of this complexity in a set of three jobs. The first two are complementary imbalanced, and identical to jobs 1 and 2 of the second scenario (Section 3.2). The third job is the basic, load-balanced job used in the first scenario (Section 3.1). The following table shows the run time results for the four algorithms.

Algorithm	Job 1	Job 2	Job 3	Turnaround	Mean response
FCFS	120	241	302	302	221
GS	185	185	305	305	225
SB	213	214	276	276	234
FCS	252	253	155	253	220

Once more, batch and gang scheduling demonstrate similar performance. Shuffling the job order so that job 3 is actually first would have improved the response time for FCFS even further, to around 181 *s*.

Another weakness of SB scheduling is exposed in this scenario: since it gives an equal treatment to all processes, fine-grained jobs suffer from the noise generated by other jobs. This is clearly shown in the performance of Job 3 under SB, which is much worse than that under FCS. Since FCS identifies the

special requirements of Job 3, it classifies it as *CS*. As such, it receives dedicated timeslots that allow it to communicate effectively, hindered only by context-switch overhead. The overall result is a decrease in turnaround time and mean response time when compared to the other algorithms.

Summary

These basic benchmarks demonstrate the inherent strengths and weaknesses of multiprogramming vs. batch schedulers, and dynamic vs. rigid schedulers. In all cases, FCS' performance was close or better to that of the scheduling algorithm most suited to each scenario, attesting to its flexibility. The next section explores how these properties are translated to two actual scientific applications.

4. Application Tests

For a more realistic experimental evaluation, we compare FCS to the other three scheduling algorithms when running two real MPI applications. The results presented in this section describe three scenarios based on different job mixes: two fine-grained applications, two medium-grain applications, and a combination of both. The applications used in this section are SWEEP3D and SAGE, both running on 128 processors.⁵ We use a multiprogramming level of 2 for these experiments, to reflect the fact that these real applications have significant memory footprints, preventing the accommodation of many instances of these applications entirely in physical memory. In addition, due to the applications' large working set and the cluster's susceptibility to noise when using 4 processors per node [20], these experiments were run with a timeslice of 200 *ms* to amortize some of the context-switching cost. Still, some overhead is noticeable for all the scheduling algorithms, and in particular with the explicit algorithms (GS and FCS).

SWEEP3D SWEEP3D [10] is a time-independent, Cartesian-grid, single-group, discrete ordinates, deterministic, particle transport code taken from the ASCI workload. SWEEP3D represents the core of a widely used method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for 50 – 80% of the execution time of many realistic simulations on current DOE

⁵These two applications are representative of the ASCI workload.

systems. SWEEP3D is characterized by very fine granularity (averaging ≈ 1 ms in the configuration tested) and a nearest-neighbor communication stencil. We used a realistic input file comprising of 10,000 cells per processor, taking ≈ 270 MB of memory per process. We increased the number of iterations from 12 to 600 so that a single run takes ≈ 111 s⁶. The combination of this data set and architecture represents a rather extreme case of a load-balanced, fine-grained application, as different configurations and architectures typically result in coarser granularities [8, 9].

We ran a simple workload consisting of two identical SWEEP3D jobs, similar to the first synthetic scenario of the previous Section, and present the results in the same format.

Algorithm	SWEEP3D 1	SWEEP3D 2	Turnaround	Mean response
FCFS	111	222	222	167
GS	232	232	232	232
SB	275	275	275	275
FCS	244	244	244	244

All the multiprogramming algorithms are hindered by the overhead associated with multitasking between highly-synchronous jobs with an active working set. Context-switch overhead, interruptions to synchronous communication, cache flushing, and skew caused by noise (and exacerbated by the fine granularity of SWEEP3D) causes a per-job slowdown of $\approx 2.25\%$ and $\approx 4.95\%$ for GS and FCS respectively, compared to batch scheduling. FCS suffers from relatively high reclassification overhead, due to the high amount of communication events. In the case of GS and FCS, there is also the occasional interruption by the NM, that is awoken to perform the context switch. While SB has the potential for less overhead, since the NM does not intervene with the scheduling, the lack of explicit coscheduling is shown to have a significant effect on SWEEP3D’s performance, with a per-job slowdown of 11.93% compared to FCFS.

SAGE SAGE (SAIC’s Adaptive Grid Eulerian hydrocode) is a multidimensional (1D, 2D, and 3D), multimaterial, Eulerian hydrodynamics code with adaptive mesh refinement (AMR) [11]. The code uses

⁶This does not change the application behavior, but merely serves to artificially increase its run time to more than a few seconds

second order accurate numerical techniques. SAGE comes from the LANL Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. SAGE has also been applied to a variety of problems in many areas of science and engineering including water shock, stemming and containment, early time front design, and hydrodynamics instability problems.

SAGE is more load imbalanced than SWEEP3D, which implies that SAGE does not always follow the classic BSP model, as the granularity varies across processes and over time. The effective granularity is thus much coarser than the computation granularity: While most processes reach a synchronization point every 5 *ms* or less, due to unequal partitioning of the input data, they are often waiting for their peers for a few more *ms*, so the average granularity over all processes is closer to 8 *ms*. Most such "computational" fragments are not large enough to fill effectively with another process, but are still large enough to classify $\approx 75\%$ of the processes as class *F*. Once more, we run two concurrent copies of SAGE with the "timing_h" input file, (using 335 *MB*/process) and present the results in the following table:

Algorithm	SAGE 1	SAGE 2	Turnaround	Mean response
FCFS	126	252	252	189
GS	260	261	261	261
SB	274	276	276	275
FCS	264	264	264	264

The different algorithms' performance in this workload shows less disparity than with SWEEP3D, indicating that SAGE lends itself rather well to multiprogramming. When comparing this workload to the previous one, the most obvious difference is the relatively coarser granularity of SAGE which lowers the performance penalty for context switching. Another mitigating factor is the occasional success of the dynamic algorithms (SB and FCS) to overlap some computation and communication of competing, load-imbalanced processes.

Combined Workload In the last part of this section we analyze a workload composed of both SAGE and SWEEP3D together, somewhat similar to the workload of the fourth synthetic scenario. This work-

load offers the opportunity to see how the two different types of jobs interact when scheduled together, which is more relevant for many supercomputing centers. Both jobs run for a similar amount of time in isolation (after adjusting SWEEP3D), and were launched together. The results are presented in the following table:

Algorithm	SAGE	SWEEP3D	Turnaround	Mean response
FCFS	126	238	238	182
GS	245	240	245	243
SB	240	272	272	256
FCS	235	224	235	230

Once more, this scenario demonstrates the adverse effect of context switch overhead on gang scheduling (compared to batch scheduling), and the significant slowdown a fine-grained application (SWEEP3D) can suffer from the lack of coordination in SB. FCS fares somewhat better than both, and slightly better than batch scheduling, since on the one hand it can coschedule SWEEP3D explicitly, and on the other make use of the small amount of load-imbalance in SAGE, to fill the computational holes. As noted in the synthetic experiments, and on other hardware configurations, when the load imbalance is higher or the context switch overhead is lower, FCS can significantly outperform the other scheduling algorithms. A situation where load imbalances are constantly created and changed due to a more realistic, dynamic workload is even more conducive to improved utilization with FCS. This scenario is discussed in the next section.

5. Dynamic Workloads

The workloads evaluated so far were relatively simple and static. Many if not most real HPC centers run more complicated workloads, with different jobs of different sizes and run times arriving dynamically to the system. With these machines, issues such as queue management, scheduling algorithms and multiprogramming levels, interactions between jobs and specific hardware configuration have significant implications. A detailed evaluation of these aspects was reported in previous work [7]. In this section

we reproduce the main results, namely, the effect of offered load on an actual cluster running a relatively long dynamic workload, under the four scheduling algorithms.

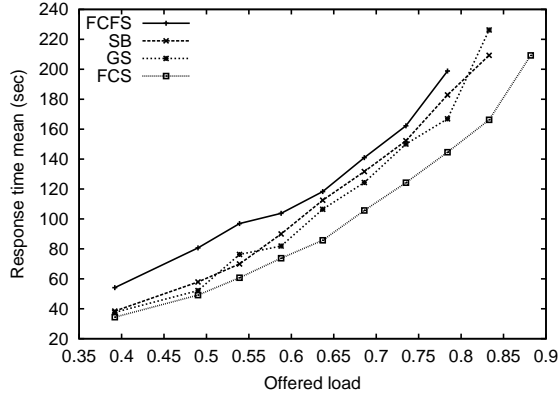
5.1 Workload and Methodology

We use a workload of 1000 jobs, whose size, run time, and arrival times are modeled using an accurate workload model [7, 15]. We use this workload as a mold for two sets of experiments, with synthetic and real applications respectively. For each set, we create a series of workloads with varying offered load by multiplying job arrival times by a constant.

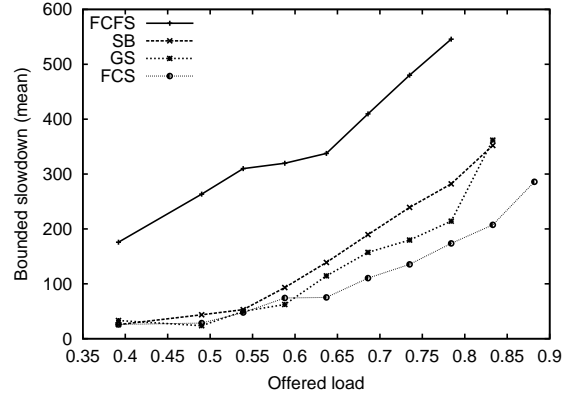
The first set runs the same synthetic application as the previous section, but with different job sizes and run times as determined by the model, and with different granularity and communication pattern, chosen randomly. The second set consists of multiple instances of SAGE and SWEEP3D with input files chosen randomly from a set of scenarios representing different physics problems. This in turn implies that we cannot control the run time of each job, only its arrival time. Moreover, there is less variety in run times in the second set. On the other hand, the applications are real and produce representative calculations, making the internal communication and computation patterns more realistic than those of the first set. We run both sets with four algorithms, FCFS with backfilling, GS, SB, and FCS. The first set, having no real memory constraints, was run with a multiprogramming level (MPL) of 6, while the second set was limited to an MPL of 2, to avoid physical memory overflow, which would require a memory-aware scheduling algorithm [18].

To estimate the offered load for each workload, we first measure the run time of each job in isolation. The offered load is then calculated as the ratio between the requested resources (sum of all jobs' run time multiplied by their sizes) by the available resources – the number of processors multiplied by the last arrival time. Accepted load cannot be measured precisely, but can be indirectly inferred from the accumulation of jobs in the queue [8]. Additionally, we stop the experiments at each algorithm's saturation point, as witnessed by a relatively constant accepted load.

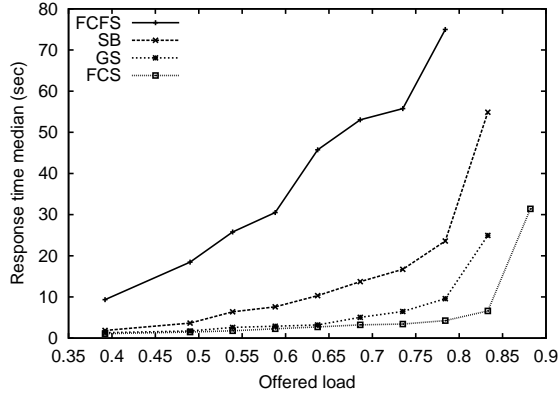
We use EASY backfilling [22] for queue management, which was shown to be very effective in lowering the average response time, in particular for short jobs and batch scheduling [16]. Since it is also



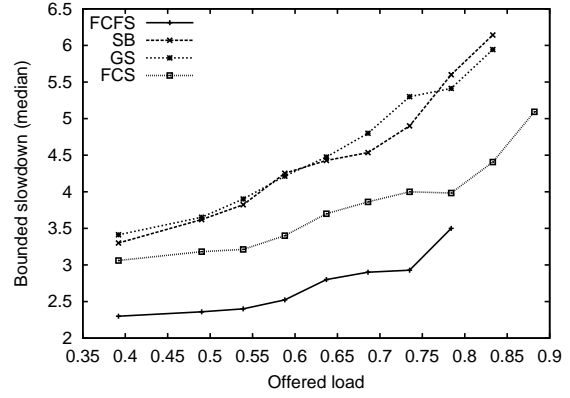
(a) Mean Response time



(b) Mean bounded slowdown



(c) Median response time



(d) Median bounded slowdown

Figure 7. Performance as a function of offered load - synthetic applications

beneficial for multiprogramming schedulers, we use EASY for them as well, multiplying the requested run time by the MPL to estimate run and reservation times [7]. The first experiment set was executed out on a different LANL cluster (*Crescendo*), using 32 Pentium-III processors (16 nodes), and the second set was executed on 64 processors of Wolverine (16 nodes).

5.2 Results and Discussion

Figures 7(a) and 7(b) show the average response time and slowdown respectively, for different offered loads and scheduling algorithms for the first set. The near-linear growth in response times with load is due to our method of varying load, by multiplying run times of jobs by a load factor. Both metrics suggest that FCS performs consistently better than the other algorithms, and FCFS seems to perform

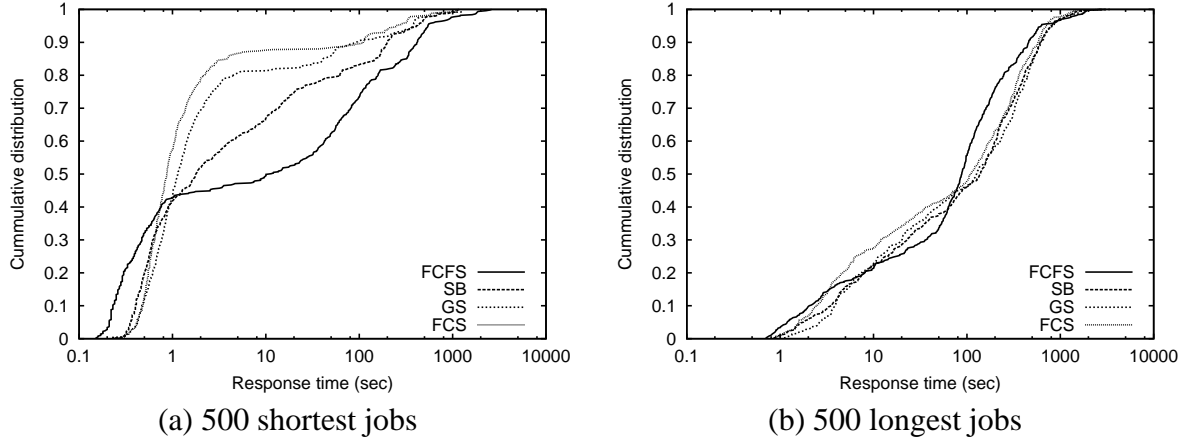


Figure 8. Cumulative distribution of response times at 74% load and FCS scheduling.

consistently worse than the others. Also, FCFS saturates at a load of $\approx 78\%$, while FCS supports a load of up to 88% in this set.

To understand the source of these differences, we look at the median response time and slowdown (Figures 7(c) and 7(d) respectively). A low median response time suggests good handling of short jobs, since most jobs are comparatively short. On the other hand, a small median slowdown indicates preferential handling of long jobs, since the lowest-slowdown jobs are mostly long jobs, that are less affected by wait time than short jobs. FCFS shows a high average slowdown and a small median slowdown. This indicates that while long jobs benefit from shorter waiting times (driving the median slowdown lower), short jobs suffer enough to significantly raise the average response time and slowdown.

To verify these biases, we look at the cumulative distribution function (CDF) of response times for the shorter 500 jobs and longer 500 jobs separately, as defined by their run time in isolation (Fig. 8). The higher distribution of short jobs with FCS attests to the scheduler’s ability to “push” more jobs toward the shorter response times. Similarly, FCFS’ preferential treatment of long jobs is reflected in Fig. 8(b).

Figure 9 shows the response time and bounded slowdown results for the second set, running SAGE and SWEEP3D. The differences between the three timesharing algorithms is not as dramatic as in the first set, mostly due to the fact that the lower MPL allows less room to express the differences between the algorithms. Still, FCS performs a little better than the other algorithms, particularly in the higher loads.

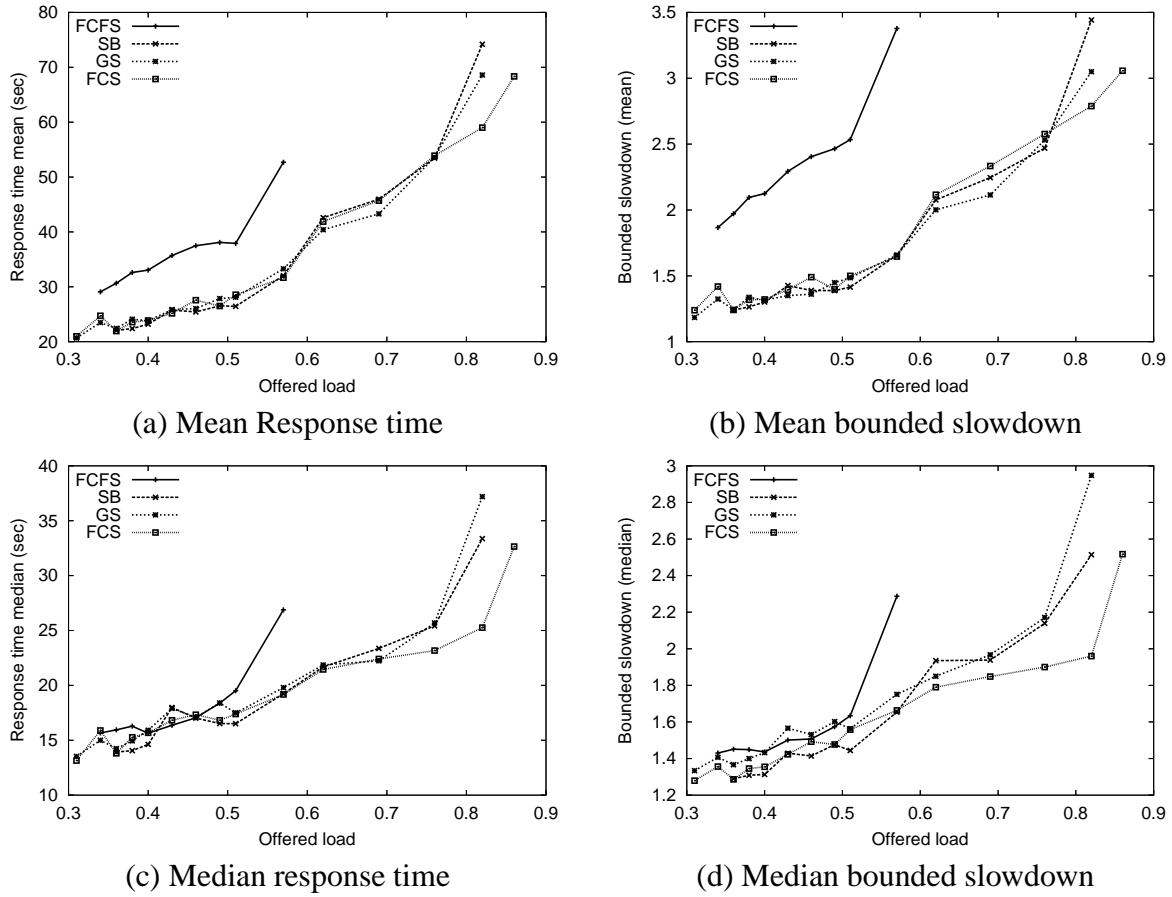


Figure 9. Performance as a function of offered load - scientific applications

FCS also saturates at an offered load higher than that of the other algorithms, while FCFS saturates at a very early point compared to the other algorithms and previous set.

We believe the reason for FCS' good performance is its ability to adapt to various scenarios that occur during the execution of the dynamic workload. In particular, FCS always coschedules a job in its first few seconds of running (unlike SB), and then classifies it according to its communication requirements (unlike GS). If a job is long, and does not synchronize frequently or effectively, FCS will allow other jobs to compete with it for machine resources. Thus, FCS shows a bias toward short jobs, allowing them to clear the system early. Since short jobs dominate the first workload, this bias actually reduces the overall system load and allows long jobs to complete earlier than with GS or SB. The opposite can be said of the FCFS scheme, which shows a bias toward long jobs, since they do not have to compete with other jobs.

6. Conclusions

An important problem with traditional parallel job scheduling algorithms is their specialization for specific types of workloads, exhibiting poor performance when the workload characteristics do not fit the model for which they were designed. For example, batch and gang scheduling perform poorly under dynamic or load-imbalanced workloads, whereas implicit coscheduling suffers from performance penalties for fine-grained synchronous jobs. Most job schedulers offer little adaptation to externally- and internally fragmented workloads, resulting in reduced machine utilization and response times.

We designed Flexible CoScheduling (FCS) to specifically alleviate these problems by dynamically adjusting scheduling to varying workload and application requirements. FCS has been fully implemented on top of STORM, and tested on three cluster architectures using both synthetic and real applications, static and dynamic workloads. The results clearly show that FCS deals well with both internal and external resource fragmentation, and is competitive with batch scheduling, gang scheduling, and implicit coscheduling. The performance advantages of FCS over other algorithms in the more realistic, dynamic workloads is rather significant: FCS saturates at an offered load approximately 16% higher than that of backfilling for synthetic applications, and 54% higher for scientific applications. The difference in saturation compared to gang scheduling and implicit coscheduling is less dramatic, but still favors FCS. In all the dynamic scenarios but one, FCS performs equally or better than the other algorithms in terms of response time and bounded slowdown.

References

- [1] Cosimo Anglano. A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations. In *Proceedings of the Ninth International Symposium on High Performance Distributed Computing*, August 2000.
- [2] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Informing Algorithms for Efficient Scheduling of Synchronizing Threads on Multiprogrammed SMPs. In *Proceedings of the International Conference on Parallel Processing*, pages 123–130, September 2001.
- [3] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, August 2001.

- [4] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, Andy B. Yoo, and Chita R. Das. Coscheduling in Clusters: Is It a Viable Alternative? In *Proceedings of SC2004*, November 2004.
- [5] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman Publishers, Inc., 1999.
- [6] Dror G. Feitelson and Larry Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–24. 1998. LNCS 1459.
- [7] Eitan Frachtenberg, Dror G. Feitelson, Juan Fernandez-Peinador, and Fabrizio Petrini. Parallel Job Scheduling under Dynamic Workloads. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.
- [8] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Dealing with Load Imbalance and Heterogeneous Resources. In *Proceedings of the International Parallel and Distributed Processing Symposium 2003, IPDPS2003*, April 2003.
- [9] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of SC2002*, November 2002.
- [10] Adolfo Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *Symposium on the Frontiers of Massively Parallel Computation*, February 1999.
- [11] Darren Kerbyson, Hank Alme, Adolfo Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of SC2001*, November 2001.
- [12] R. Kettimuthu, V. Subramani, S. Srinivasan, T. B. Gopalsamy, D. K. Panda, and P. Sadayappan. Selective Preemption Strategies for Parallel Job Scheduling. In *International Conference on Parallel Processing*, August 2002.
- [13] JunSeong Kim and David J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 202–216, February 1998.
- [14] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, LNCS 1291, pages 215–237, 1997.
- [15] Uri Lublin and Dror G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel & Distributed Computing*, 63(11):1105–1122, November 2003.
- [16] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

- [17] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.
- [18] Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos. Adaptive Scheduling under Memory Constraints on Non-Dedicated Computational Farms. *Future Generation Computer Systems*, 19(4):505–519, May 2003.
- [19] Fabrizio Petrini, Wu chun Feng, Adolfo Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.
- [20] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of SC2003*, November 2003.
- [21] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, LNCS 1459, pages 231–256. 1998.
- [22] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective Reservation Strategies for Backfill Job Scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, LNCS 2357, pages 55–71. 2002.
- [23] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [24] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, June 2003.
- [25] ASCI Technology Prospectus: Simulation and Computational Science. Technical Report DOE/DP/ASC-ATP-001, National Nuclear Security Agency, July 2001.