Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources*

Eitan Frachtenberg[1,2], Dror G. Feitelson[2], Fabrizio Petrini[1] and Juan Fernandez[1]

[1]CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory (LANL)
{eitanf,fabrizio,juanf}@lanl.gov

[2]School of Computer Science and Engineering
The Hebrew University, Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Fine-grained parallel applications require all their processes to run simultaneously on distinct processors to achieve good efficiency. This is typically accomplished by space slicing, wherein nodes are dedicated for the duration of the run, or by gang scheduling, wherein time slicing is coordinated across processors. Both schemes suffer from fragmentation, where processors are left idle because jobs cannot be packed with perfect efficiency. Obviously, this leads to reduced utilization and sub-optimal performance. Flexible coscheduling (FCS) solves this problem by monitoring each job's granularity and communication activity, and using gang scheduling only for those jobs that require it. Processes from other jobs, which can be scheduled without any constraints, are used as filler to reduce fragmentation. In addition, inefficiencies due to load imbalance and hardware heterogeneity are also reduced because the classification is done on a per-process basis. FCS has been fully implemented as part of the STORM resource manager, and shown to be competitive with gang scheduling and implicit coscheduling.

Keywords: *Cluster computing, load balancing, job scheduling, gang scheduling, parallel architectures, heterogeneous clusters, STORM*

1. Introduction

Workstation clusters are steadily increasing in both size and number. 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 [3]. Load imbalance can have a marked detrimental effect on many parallel programs. A large part 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 [17, 5, 10]. 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.

Load imbalance has two main sources: application 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. This uneven taxing of resources creates a situation where some parts of the parallel program run slower than others, and a load imbalance occurs [7].

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. For example, this may happen in computing centers that buy processing nodes over a period of time, thus taking advantage of technological improvements, or in grid computing.

The traditional approach to this problem is to tackle application load imbalance at the application level: the programmer tries to balance the resources by changing the structure of the parallel program. This approach is usu-

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

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

ally time consuming and yields diminishing returns after an initial phase of code restructuring and optimizations; in fact, there are 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 [2] 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 system 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. 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) [1] can potentially alleviate load imbalance and increase resource utilization. However, ICS is not always able to handle all job types because it cannot rely on global coordination.

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 try to detect possible load imbalances, and (3) classification of the applications according to these metrics. On top of this, we propose a coscheduling mechanism that uses this application classification to make scheduling decisions. The scheduler attempts to coschedule processes that would most benefit from it, while scheduling other processes to increase overall system utilization and throughput. A specific application that suffers from load imbalances will not complete faster with this scheduler compared to other schedulers. Obviously, any given application gets the best service when running by itself on a dedicated set of nodes, as when running in batch mode. But this can block other jobs. The proposed scheduler will prevent each job from wasting too many system resources, and the overall system efficiency and responsiveness will be improved.

We demonstrate this methodology with a streamlined implementation on a resource manager, called STORM [6]. 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. 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.

Finally, we provide an extensive experimental evaluation which ranges from simple workloads that provide insights on several job scheduling algorithms to experiments with real applications representative of the Accelerated Strategic Computing Initiative (ASCI) workload. We evaluate FCS by running different mixes of test programs on a 32-node/64-processor cluster using different schedulers. Specifically, we compare the performance achieved by FCS with that of first-come-first-served batch scheduling (FCFS), gang scheduling (GS), local scheduling with busy waiting, and local scheduling with spin blocking (SB), which is very similar to ICS [1] in this context. Different schedulers provide the best performance for different job mixes. However, in all the cases tested, FCS provided essentially the same performance as the best other scheduler. This testifies to its flexibility, its ability to identify the characteristics of the workload applications, and its ability to make good scheduling decisions based on this identification.

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 heterogeneous hardware or software, by using dynamic measurement of applications' communication patterns and classification of applications into distinct types. FCS is implemented on top of STORM [6], which allows both for global synchronization through scalable global context switch messages (heartbeats) and local scheduling by a daemon run on every node, based on their locally-collected information.

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.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

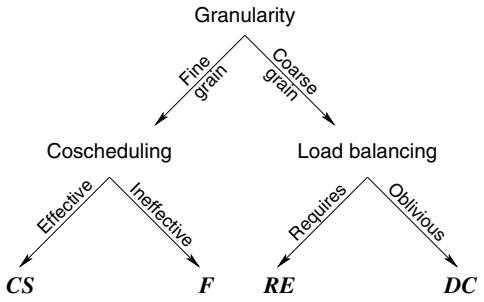


Figure 1. Decision tree for process classification

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 can also define another class, called *RE* (rate-equivalent), for jobs that have little synchronization, but require a similar amount of CPU time for all their processes. However, detection of *RE* processes cannot be made in run-time with local information only, so 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 timeslice.[1] 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's communication operations, 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 read them. Only synchronous (blocking) communication calls are monitored, since non-blocking communications do not require tight synchronization and should not affect scheduling (thus a call to MPI_Isend(), for example, is non-blocking, but MPI_Wait() is considered blocking).

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

This classification differs in two important ways from the one suggested by Lee et al. [11]. 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 [18], which is based on CPU utilization and is done at the job rather than the process level.

2.2. Scheduling

The principles behind scheduling in FCS are as follows:

- *CS* processes should be coscheduled and should not be preempted.
- *F* processes should be coscheduled but can be preempted when synchronization is not effective.
- *DC* processes impose no restrictions on scheduling.

The infrastructure used to implement this scheduling algorithm is based on the implementation of conventional gang scheduling. 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 managers (NM) instructing them to switch from one slot to another. A crucial characteristic is that the node managers do not have to comply: they are free to overrule the MM's directives based on their local measurements and classifications.

Algorithm 1 shows the behavior of the node manager upon receipt of a multi-context-switch message (note that this is done independently on each node, without global coordination). 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 that *F* processes create because of their synchronization problems. An *F* process that waits for pending communication should not block immediately, but rather spin for some time to avoid unnecessary context switch penalties, as in ICS [1].

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 [16] and ICS in that:

1. A *CS* process in FCS cannot be preempted before the time slot expires even if an incoming message arrives

[1] In 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 might 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".

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

Algorithm 1: Context switch algorithm for FCS

```
// context_switch: switch from one process to another process
// Invoked on each processing node 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 all 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, make sure it has
         high enough priority over all other processes to
         ensure that it will run uninterrupted.
    end
end
```

for another process (processes classified as *CS* have shown that it is not worthwhile to deschedule them in their time slot, due to their fine-grain synchronization). It should not yield the processor on blocking events until its timeslice expires.

2. The local scheduler's decision in choosing among processes in the *DC* time slots and *F* gaps is affected by the communication characterization of processes, which could lead to less-blocking processes and higher utilization of resources.

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 typically a class change, but can also be triggered by process age.
2. Parameters measured or determined by the scheduling layer, also detailed in Table 1.
3. Algorithm constants, shown in Table 2, together with the values we used for the experiments.

Measurements are taken whenever a process is scheduled to run. 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 their progress, except possibly for *F* processes that get a higher priority in their slot. In these cases, it is used mainly to track the age of a process using the $cslots$ and $tslots$ counters.

	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

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. 25ms was shown to be a good choice for these considerations in [6].
- T_{spin} was chosen to be high enough to accommodate most communication operations (with the network used in our experimental evaluation, these typically complete in few tens of microseconds [13]), and low enough so that resources are not wasted unnecessarily.
- $cslots_{MIN}$ should allow enough time for some initial-

Name	Description	Value
T_{slice}	Timeslice quantum	25ms
T_{spin}	Spin time for spin-block communications	100μs
$cslots_{MIN}$	Minimum value of $cslots$ for process to be evaluated for a class change	10
DC_{thresh}	*DC* granularity threshold: above this value process is *DC*	1s
CS_{thresh}	*CS* granularity threshold: below this value process is *CS*	10ms
F_{thresh}	threshold of computation granularity to identify processes waiting for communication as *F*	$0.75 \times CS_{thres}$
$tslots_{MAX}$	Maximum value of $tslots$, after which a reset to class CS is forced	16384

Table 2. FCS constants and values used in experiments.

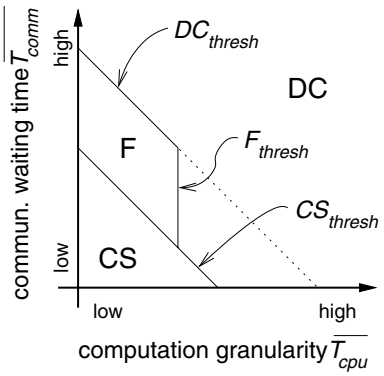


Figure 2. Phase diagram of classification algorithm

izations 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 10ms or so (see below in Section 3).

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 timeslice, so this happens at deterministic, predictable times throughout the machine. In this way, if the time has arrived to reset a process to class *CS*, it is guaranteed that all the processes of the same job will be reset together (otherwise they might not really be coscheduled).

The algorithm can be explained with the aid of 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 time and the average communication time. Therefore constant granularity is represented by diagonals from upper left to lower right. *CS* processes are those that 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 [6], 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. The basic software architecture is a set of daemons, one for the whole 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 addition to MPI to measure and export information on processes' synchronous communication to the NM, and a module in the NM that can translate this information into a classification, and schedule processes based on their class.

2.6. Implementation Issues

Measuring process statistics can be both intrusive and imprecise if not done carefully. It is important to perform the 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 has been verified using synthetic applications for which the

Algorithm 2: Classification function for FCS

```
// re-evaluate, and possibly re-classify the process
// using FCS parameters and measurements
procedure FCS_reclassify
begin
    if tslots mod tslots_MAX == 0    //  Time for a reset:
        reset_process and return // Changes class back to CS
    if cslots < cslots_MIN
        return             // Not running long enough in current class
    if g > DC_thresh
        class = DC         // Coarse granularity
    else if g < CS_thresh
        class = CS         // Fine granularity
    else if T_cpu < F_thresh
        class = F          // Communication too slow
    else
        class = DC        // Communicates well but infrequently
end
```

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, so that the scheduler, which is a different process, can access them without issuing a system call. While this mechanism is asynchronous and there could be a lag between the actual communication event and the time the scheduler gathers the information, these parameters converge quickly to provide an accurate picture of the process characteristics.

For counting communication events (C_{comm}), we employed the following heuristic: multiple communication 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. In our implementation, the finest granularity that can be detected is $\approx$ 2ms, although this value can be changed by modifying the Linux HZ constant.

Another measurement issue is the time the process spends spinning while waiting for communication to terminate. This time is accounted for and subtracted from T_{cpu}, since it can effect the precision for fine-grained jobs.

3. Experimental Results

This section presents the experimental results comparing the performance of FCS to that of four other scheduling algorithms. Two baseline algorithms are first-come-first-served (FCFS) and local scheduling, which represent two extremes: a completely dedicated job assignment versus a completely shared, uncoordinated one. We also compare to gang scheduling (GS) and spin-block (SB). SB is very similar to implicit coscheduling (ICS) [1], and represents an effective way to time-share a machine without global coordination as in gang scheduling. With SB, processes that wait for synchronous communication poll for a given interval, and only if the communication has not completed by this time they block (in contrast, gang and locally-scheduled processes always busy-wait). In this way, processes tend to self-synchronize across the job, so relatively good coordination is achieved without the need for a global mechanism. In ICS the spin time can be adaptive, thus decreasing inefficiencies resulting from spinning too long. In our implementation of SB we chose a small constant time for spinning (100μs), so that very little time is wasted. Note that typical communication operations with the Quadrics interconnect complete in far less than this time ($\approx 10 - 50\mu$s), so if two communicating processes are coscheduled, they are almost guaranteed to complete the communication within this time interval.

3.1 Experimental Setup

The hardware used for the experimental evaluation was the "crescendo" cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550), and a 128-port Quadrics switch [13] (using only 32 of the 128 ports). Each compute node has two 1 GHz Pentium-III processors, 1 GB of ECC RAM, two independent 66MHz/64-bit PCI buses, a Quadrics QM-400 Elan3 NIC [13, 14, 15] 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. We further modified the kernel by changing the default HZ value from 100 to 2048. This has a negligible effect on operating system overhead,

but makes the Linux scheduler re-evaluate process scheduling every $\approx 500\mu$s. As a result scheduling algorithms (in particular SB and Local) become more responsive [4].

3.2 Verification Tests

In this section we analyze the behavior of FCS under various synthetic benchmarks, and compare it to the other four scheduling algorithms. For each scenario, we describe its setup, show the run time of each job in the workload and the total turnaround time, and analyze the results. The best turnaround time (from the launch of the first job to the end of the last) in each table is shown in boldface.

All scenarios use a simple workload of 2-4 jobs with the same arrival time, using only 2 of the cluster's nodes. The basic "building-block" job has four processes (running on two nodes), communicating in a ring pattern every 5ms, which was chosen to be a granularity fine enough to be representative of scientific applications [8]. Running such a job in isolation takes approximatively 60s, but could vary by up to 0.1% due to noise. The benchmarks were run several times and produced relatively small variations in results (typically less than 1s in total runtime).

3.2.1 Fine-grained jobs

In the first scenario, we run two identical jobs concurrently. Both jobs are fine-grained, requiring to be coscheduled to communicate effectively. The following table presents the results for running this workload, giving the termination time in seconds for each job and for the complete set:

Algorithm	Job 0	Job 1	Max
FCFS	60.00	119.95	119.95
Local	234.79	230.95	234.79
GS	118.08	118.06	**118.08**
SB	125.36	125.38	125.38
FCS	118.34	118.39	118.39

Since fine-grained, balanced jobs require a dedicated environment to proceed effectively, FCFS scheduling and GS offer the best performance. Remarkably, GS even outperforms FCFS by a small margin. This is the result of a limited amount of overlap between the computation of one job and the communication of the other, which happens right after a context switch, as communication is handled by the Elan NIC independent of what the CPU is doing. Both local and SB scheduling perform poorly in this scenario, due to the lack of global coordination (SB actually performs much worse in relative terms as the granularity becomes finer). 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 higher than that of GS, due to the added overhead of process classification.

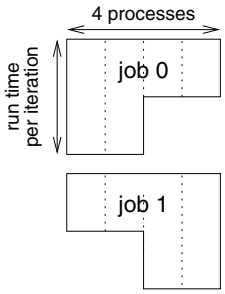


Figure 3. Test scenario of two load-imbalanced jobs

3.2.2 Load-imbalanced jobs

This scenario represent a simple load-imbalance case with two complementing jobs, as seen in Figure 3. Processes 0 and 1 (using 0-based counting) of job 0 compute twice as much per iteration as processes 2 and 3, while for job 1 the situation is reversed. The faster processes compute the same amount as in the previous scenario.

Algorithm	Job 0	Job 1	Max
FCFS	116.57	233.61	233.61
Local	301.82	300.79	301.82
GS	231.36	231.91	231.91
SB	177.86	179.49	179.49
FCS	176.26	177.64	**177.64**

This scenario exposes the inefficiency in running load-imbalanced jobs in dedicated mode. 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% (the ratio of total runtime is not exactly 2:1, since the computation represents only a part of the total runtime. Communication time remains largely unchanged). Local scheduling performs poorly, because the jobs are fine-grained. 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 classifies the first job's processes as *DC*, *DC*, *F* and *F* respectively, and the second job's as *F*, *F*, *DC*, and *DC*. 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 maximum resource utilization: both jobs complete after running for 150% of the time

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

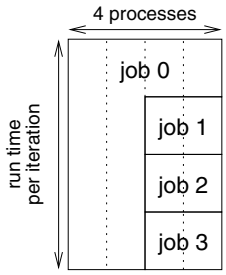


Figure 4. Test scenario of four complementing jobs

it took the previous scenario, which corresponds to the new amount of work.

3.2.3 Complementing jobs

This scenario demonstrates the ability of various algorithms to pack jobs efficiently in an extremely imbalanced workload. It consists of one four-process job and three two-process jobs running on PEs 2 and 3 (see Figure 4). All the jobs running on PEs 2 and 3 compute the same basic amount as in the previous scenarios, but processes 0 and 1 of the first job compute four times as much per iteration. An optimal scheduler should pack all these jobs so that the total turnaround time does not exceed that of the first job when run in isolation. This packing is shown in the figure.

Alg.	Job 0	Job 1	Job 2	Job 3	Max
FCFS	231.25	58.99	59.60	58.97	408.26
Local	356.14	233.13	233.58	233.73	356.14
GS	404.72	232.11	232.21	232.19	404.72
SB	261.15	229.20	229.22	229.22	261.15
FCS	236.33	233.44	233.54	231.96	**236.33**

Once more, FCFS and GS exhibits similar turnaround time — the combined time of all the jobs run in isolation. Local scheduling does slightly better, since the large 'holes' created by job 0 on PEs 2 and 3 are partially filled with the other jobs, at the expense of job 0's turnaround time. SB shows some ability to load-balance the jobs, but since it lacks a detailed knowledge of the processes requirements, it can only go so far — Job 0 still takes about 13% more time to run than it should under an optimal scheduler (decreasing further in performance as the load-imbalance grows). FCS classifies all the processes as *DC*, except for the processes 2 and 3 of job 0 which are classified as *F*. As such, they receive priority in their time slot, and thus the total runtime of job 0 is hardly affected by the existence of other jobs, which pack neatly into the other timeslices. In fact, the total turnaround time of this workload with FCS is within 2% of the optimal value of 231s.

From this point on, local scheduling will no longer be considered since it does not provide any advantage over the other algorithms.

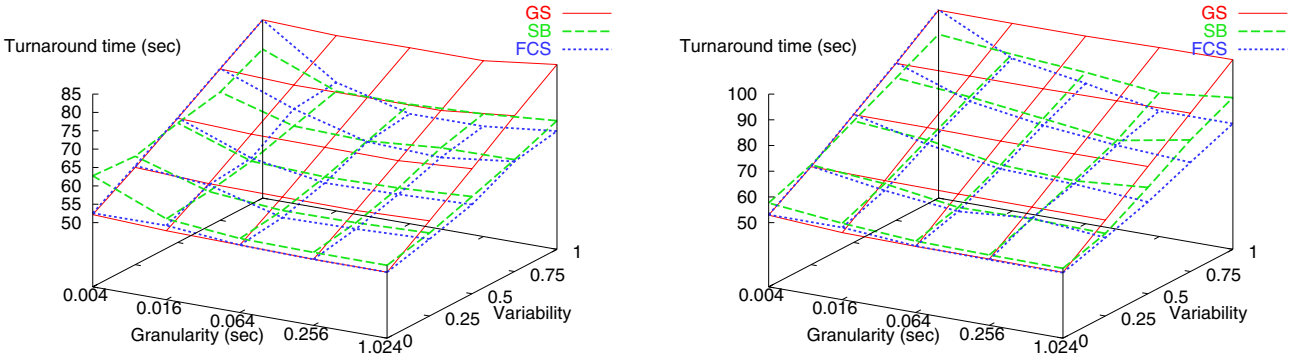
3.3 Exploring the Parameter Space

In another set of experiments we performed a comprehensive survey of the relevant parameter space, defined by

- Computation granularity from 4ms to 1.024s.
- Variability from 0 to $\pm 100\%$, measured relative to the granularity.
- Scheduling algorithm (GS, FCFS, FCS, and SB).
- Communication pattern: nearest neighbor grid (NN) and all-to-all, represented by a barrier.

In all cases, 4 identical jobs were executed on all 32 nodes/64 processors. The results are shown in Figure 5. The main points can be summarized as follows:

- GS and FCFS exhibit similar performance, with linear degradation as variability grows, because idle time is simply wasted. To reduce clutter, FCFS results are not shown in the graphs.
- SB and FCS can make use of idle time by scheduling other processes. Therefore their degradation with variability is much less pronounced. Both also improve as the granularity grows.
- FCS converges to GS and FCFS for the very fine granularity, regardless of variability. This is probably because it classifies all processes as CS, and thus degenerates to GS. Additional measurements in which the variability was increased up to 200% indeed showed that FCS begins to perform better than GS and FCFS when the total granularity surpasses the CS_{thresh} threshold.
- For very fine granularity, SB is better than the other schemes when the variability is high, but worse when it is low. The reason that it performs better than FCS for low granularity and high variability is that FCS insists on gang scheduling unnecessarily as explained above. This implies that our threshold for classifying as *CS* may be too high, as it is possible to benefit from the variability for finer granularities. The reason why SB is not so good with fine granularity and low variability is that it wastes time spinning for unscheduled processes; in the other schemes, jobs are coscheduled so this does not happen. At high granularity and variability FCS manages to use idle resources better than SB.
- For 0 variability, SB converges to FCFS, and FCS converges to GS, which is marginally better.



Nearest Neighbor

Barrier

Figure 5. Results for exploration of the parameter space

3.4 Applications Testing

This section presents results for two scenarios based on real applications: one composed of fine-grained and balanced jobs and another with load-imbalanced jobs. The applications used in this section are SWEEP3D and SAGE. They were run on the entire cluster, using all 32 nodes/64 processors.

SWEEP3D [8] 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 systems. SWEEP3D is characterized by a fine granularity and a nearest-neighbor communication stencil. In the configuration tested each compute step takes ≈ 3.5ms, and the total runtime of the application is ≈ 48s. Our workload consists of four concurrent copies of SWEEP3D with the same input data set. Since results can vary by a few percents, we ran each workload several times and computed the median of those. The total turnaround time for this workload and different scheduling algorithms is shown in the following table:

Algorithm	Max
FCFS	**193.03**
GS	194.57
SB	208.47
FCS	197.49

Just as for the synthetic application of Section 3.2.1, we can see that FCFS and GS perform similarly, providing optimal performance for fine-granularity jobs. FCS performs within 2% of these algorithms, paying a slight performance hit for the classification overhead. SB is the slowest of the lot, since it has no global coordination for such jobs. While the performance difference in not very large in this case, we may expect the gap to grow as granularity decreases for other configurations.

SAGE (SAIC's Adaptive Grid Eulerian hydrocode) is a multidimensional (1D, 2D, and 3D), multimaterial, Eulerian hydrodynamics code with adaptive mesh refinement (AMR) [9]. The code uses second order accurate numerical techniques. SAGE comes from the Los Alamos National Laboratory 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.

The test workload consisted of three copies of the program, but with three different input files, representing different run times and load-imbalances. The following table shows the runtime of each job under the different algorithms:

Algorithm	Job 0	Job 1	Job 2	Max
FCFS	39.24	125.36	220.16	220.16
GS	120.41	222.03	227.02	227.02
SB	124.22	189.95	200.46	**200.46**
FCS	112.9	194.95	205.81	205.81

Once more we can observe FCS's ability to interleave load-imbalanced jobs to improve system utilization and overall job run time. It performs nearly as well as SB, and noticeably better than FCFS and GS, which cannot use the CPU-time gaps created by the imbalanced jobs.

In general, we can see that FCS's advantage for real

workloads is that it adapts well to different applications' requirements and idiosyncrasies. FCS performs at near-optimal performance and utilization in various kinds of scenarios, whereas existing algorithms are mostly tuned for specific types of applications.

4. Conclusions and Future Work

Flexible coscheduling is designed to alleviate the inefficiencies of gang scheduling. These include problems of fragmentation, when jobs do not pack together to utilize all processors, problems of load imbalance, where processes in the same job place different loads on the processors, and problems of heterogeneity, where processors do not provide the same level of support to different processes. The solution is based on a classification of the processes according to their needs and behavior, and dynamic scheduling based on this classification. In particular, processes that do not need or benefit from gang scheduling are simply not gang scheduled, enabling more flexible and efficient use of the processors.

FCS has been fully implemented on top of STORM, and tested on a 32-node/64-processor system using both synthetic and real applications. The results indicate that it is competitive with FCFS scheduling, gang scheduling, local scheduling, and local scheduling with spin-block synchronization (similar to implicit coscheduling). In particular, different test scenarios expose different strengths and vulnerabilities of the other schedulers. FCS was always either the best performer or very close to the best.

In future work we intend to continue the development and testing of FCS. One idea that has not yet been implemented is to use dynamic spinning times for synchronization. We are also planning to implement BCS [12] on top of STORM, and compare the performance and benefits of FCS relative to BCS. In addition, we plan to execute more tests with more varied workloads.

Acknowledgements

The authors would like to thank Kei Davis for many helpful technical comments that enhanced the readability of the paper.

References

[1] A. C. Arpaci-Dusseau, "*Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*". *ACM Trans. Computer Systems* **19(3)**, pp. 283–331, Aug 2001.

[2] *ASCI Technology Prospectus: Simulation and Computational Science*. Technical Report DOE/DP/ASC-ATP-001, National Nuclear Security Agency (NNSA), Jul 2001.

[3] D. E. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman Publishers, Inc., 1999.

[4] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, Jun 2003.

[5] D. G. Feitelson and L. Rudolph, "*Metrics and Benchmarking for Parallel Job Scheduling*". In *Job Scheduling Strategies for Parallel Processing*, LNCS vol. 1495, pp. 1–24, Springer-Verlag, Mar 1998.

[6] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll, "*STORM: Lightning-Fast Resource Management*". In *Supercomputing 2002*, Nov 2002.

[7] A. Hoisie, D. J. Kerbyson, S. Pakin, F. Petrini, H. J. Wasserman, and J. Fernandez-Peinador, *Identifying and Eliminating the Performance Variability on the ASCI Q Machine*. Technical Report LA-UR-03-0138, Los Alamos National Lab, Jan 2003.

[8] A. Hoisie, O. Lubeck, and H. Wasserman, "*Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters*". In 9th *Symp. Frontiers of Massively Parallel Computation*, Feb 1999.

[9] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings, "*Predictive Performance and Scalability Modeling of a Large-Scale Application*". In *Supercomputing 2001*, Nov 2001.

[10] J. Kim and D. 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*, pp. 202–216, Feb 1998.

[11] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "*Implications of I/O for Gang Scheduled Workloads*". In *Job Scheduling Strategies for Parallel Processing*, LNCS vol. 1291, pp. 215–237, Springer-Verlag, Apr 1997.

[12] F. Petrini and W. Feng, "*Buffered Coscheduling: a New Methodology for Multitasking Parallel Jobs on Distributed Systems*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 439–444, May 2000.

[13] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "*The Quadrics Network: High Performance Clustering Technology*". *IEEE Micro* **22(1)**, pp. 46–57, Jan-Feb 2002.

[14] Quadrics Supercomputers World Ltd., *Elan Programming Manual*. Jan 1999.

[15] Quadrics Supercomputers World Ltd., *Elan Reference Manual*. Jan 1999.

[16] P. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "*Dynamic Coscheduling on Workstation Clusters*". In *Job Scheduling Strategies for Parallel Processing*, LNCS vol. 1459, pp. 231–256, Springer-Verlag, Mar 1998.

[17] L. G. Valiant, "*A Bridging Model for Parallel Computation*". *Comm. ACM* **33(8)**, pp. 103–111, Aug 1990.

[18] Y. Wiseman and D. G. Feitelson, "*Paired Gang Scheduling*". *IEEE Trans. Parallel & Distributed Systems*, 2003. (to appear).