Symbiotic Space-Sharing on SDSC's DataStar System

Jonathan Weinberg and Allan Snavely

San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093-0505, USA
{jonw,allans}@sdsc.edu,
http://www.sdsc.edu

Abstract. Using a large HPC platform, we investigate the effectiveness of "symbiotic space-sharing", a technique that improves system throughput by executing parallel applications in combinations and configurations that alleviate pressure on shared resources. We demonstrate that relevant benchmarks commonly suffer a 10-60% penalty in runtime efficiency due to memory resource bottlenecks and up to several orders of magnitude for I/O. We show that this penalty can be often mitigated, and sometimes virtually eliminated, by symbiotic space-sharing techniques and deploy a prototype scheduler that leverages these findings to improve system throughput by 20%.

1 Introduction

On SDSC's DataStar [3], as on all parallel systems, processes must share resources. Because the system does not time-share, each process receives its own processor with a dedicated level 1 cache. However, two processors must share a level two cache. The eight processors on each node must share a level 3 cache, main memory, an on-node file system, and bandwidth to off-node I/O.

Sharing, by its very nature, entails compromise. In the realm of parallel processing, that compromise may lead to performance degradation. The more heavily coexisting processes make use of a shared resource, the more likely it is that the performance of that resource will suffer. Heavy use of a shared cache might lead to lower hit rates, and consequently, lower per-processor throughput. As more processes make simultaneous use of a shared I/O system, blocking times increase and performance degrades.

Because the consequences of resource sharing are often ill-understood, scheduling policies on production space-shared systems avoid inter-job sharing wherever possible. On DataStar, for instance, nodes are never time-shared and parallel jobs have exclusive use of the nodes on which they run. Even then, the system's General Parallel File System (GPFS) remains a shared resource among all running jobs.

This is not an ideal policy in several circumstances. Resource utilization and throughput suffers when small jobs are forced to occupy an entire node while making use of only a few processors. The policy also encourages users to squeeze large parallel jobs onto the fewest number of nodes possible since doing otherwise is both costly and detrimental to system utilization. Such configurations are not always optimal; the processes of parallel jobs often perform similar computations, consequently stressing the same shared resources and exacerbating the slowdown due to resource contention.

In such situations, a more flexible and intelligent scheduler could increase the system's throughput by more tightly space-sharing *symbiotic*[1] combinations of jobs that interfere with each other minimally. Such a scheduler would need to recognize relevant job characteristics, understand job interactions, and identify opportunities for non-destructive space-sharing.

The purpose of this study is to investigate the feasibility of such an approach and quantify the extent to which it could improve throughput if implemented. To address these questions, we must determine:

* To what extent and why do jobs interfere with themselves and each other?
* If this interference exists, how effectively can it be reduced by alternative job mixes?
* Are these alternative job mixes feasible for parallel codes and what is the net gain?
* How can a job scheduler create symbiotic schedules?

We explore each of these questions in sections 3 through 5 respectively. This discussion is preceded by details of our hardware environment in section 2 and succeeded by comments on related and future work in sections 7 and 8.

2 Hardware Environment

The results described in this paper were derived from application runs on the San Diego Supercomputer Center's DataStar. The machine contains 272 IBM P655+ nodes, each consisting of 8 Power4 processors. Of those nodes, 171 are composed of 1.5 GHz processors while the others 1.7. Only the former were utilized for this study.

Each POWER4 processor contains a 32 KB L1 data cache. Two processors together comprise a *chip* and share a 1.5 MB L2 cache. The L3 cache on each chip is combined with that on the others to create a single node-wide, address-interleaved L3 cache of 128 MB.

Each node is also equipped with 16 GB of memory and a local scratch file system of approximately 64GB. Nodes are directly connected to the GPFS (IBM's parallel file system) through a Fibre Channel link and to each other by the Federation interconnect.

DataStar schedules jobs using a batch queueing model implemented by LoadLeveler [12]. Because the scheduler interface does not allow users to directly request that jobs be coscheduled, we achieved this effect when necessary by deploying MPI jobs that execute the desired sub-jobs on specified processors depending on rank.

3 The Effects of Sharing Resources

As an initial starting point, we can broadly divide the resources shared by processors on DataStar's nodes into two categories: memory and I/O. The memory resources consist of the three levels of cache along with the node's 16GB of main memory. The I/O resources consist of the on-node file system along with bandwidth to the system's GPFS.

[1] *Symbiosis* is a term borrowed from Biology meaning the graceful coexistence of organisms in close proximity. We generalize the term to co-scheduled processes and emphasize the form in which neither does harm to the other.

To gauge the performance effects of resources sharing, we run a set of single-processor benchmarks meant to stress each resource. We then measure the slowdown incurred by each benchmark as we increase the number of its instances running concurrently on a single node. The maximum slowdown is displayed. When we refer to N concurrent instances of a benchmark, we refer to N independent, single-processor runs of some benchmark running concurrently on a single node. We calculate slowdown as $(T_N - T_1)/T_1$ where T_i is the runtime of the benchmark while i instances of it run concurrently on the node.

3.1 Memory Sharing

To test performance degradation of the memory subsystem, we choose the following three benchmarks, each meant to stress different sections of the system:

GUPS - Giga-Updates-Per-Second measures the time to perform a fixed number of updates to random locations in main memory [8, 1]. We use it to investigate the effects of high demand on main memory bandwidth.

STREAM - A simple synthetic benchmark that measures sustainable memory bandwidth for vector compute kernels, commonly encountered in high-performance computing, by performing a long series of short, regularly-strided accesses through memory [8, 2]. STREAM is highly cacheable and prefetchable and we therefore use it stress the machine's cache structure.

EP - Embarrassingly Parallel is one of the NAS Parallel Benchmarks [6]. It evaluates an integral by means of pseudorandom trials and is a compute-bound code. We use this as a control group to discern between performance degradation in the other benchmarks due to resource sharing and that attributable to other overheads.

Figure 1 shows the slowdown of each type of application. EP appears only slightly sensitive to the number of concurrent instances running on the node while GUPS and STREAM show a slowdown of up to 18% and 30% respectively.

Of note is the non-linear increase in slowdown. Since the majority of the slowdown is caused by the latter instances, we can speculate that running symbiotic jobs on those processors has the potential to eliminate a disproportionate share of the performance degradation.

The large jump in slowdown caused by adding a fifth instance of GUPS is particularly indicative of resource sharing. When fewer than five processes run on the node, DataStar is able to spread them onto separate chips and minimize resource sharing. Once a fifth instance of GUPS is added however, at least two processes must share one of the four chips and consequently an L2 cache. When sharing the L2 cache, each processes receives degraded service from it. Table 1 shows the L2 miss rates of each processor as more instances of GUPS are added. When two processes cohabitate a single chip, the miss rate increases from around .62 to .78, causing the sharp drop in performance.

To verify that our observations from these benchmarks are representative of other applications and can be generalized, we repeat part of this experiment using single-processor runs of the NAS Parallel Benchmarks [6]. We use version 3.2 and problem sizes of class B. None of these benchmarks performs any significant I/O.

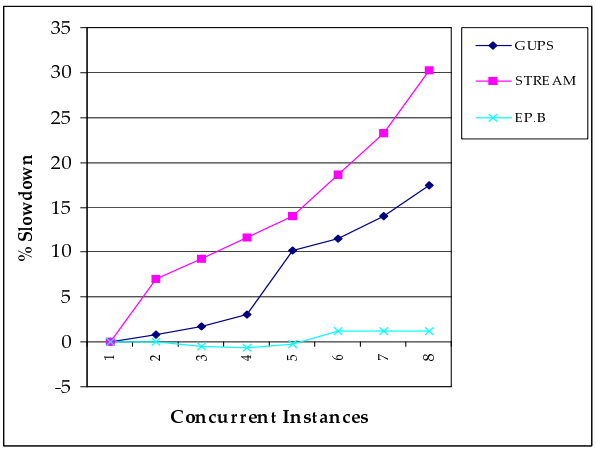

Fig. 1. Slowdown of memory intensive benchmarks as more instances of each run concurrently on a single node.

Chip 0		Chip 1		Chip 2		Chip 3	
P0	P4	P1	P5	P2	P6	P3	P7
61	-	-	-	-	-	-	-
.61	-	.61	-	-	-	-	-
.61	-	.61	-	.61	-	-	-
.62	-	.62	-	.62	-	.62	-
.76	.76	.60	-	.60	-	.60	-
.76	.76	.76	.76	.60	-	.60	-
.78	.78	.78	.78	.78	.78	.67	-
.78	.78	.78	.78	.78	.78	.78	.78

Table 1. L2 miss rates as more processors of a node run GUPS concurrently.

The results in Table 2 confirm that slowdown from memory subsystem sharing tends to fall in the range of 10-60% and that the majority of performance degradation is often resultant of using the second half of a node. Further, we notice that the slowdown incurred by each benchmark due to the first four instances varies minimally, generally within 5%.

APP	BT	MG	FT	DT	SP	LU	CG	IS
4P	8	15	1	20	20	16	14	14
8P	12	48	30	38	33	41	54	58

Table 2. Percent Slowdown of NPB while 4 and 8 instances of each run concurrently on a node.

3.2 I/O Sharing

To extend our investigation to shared I/O resources, we repeat the experiments from Section 3.1 using I/O Bench [4], a synthetic benchmark that measures the rate at which a machine can perform reads and writes to disk. The benchmark performs a series of sequential, backward, and random read and write tests.

We configure I/O Bench to use a file size of 600MB and block size of 4K. Each benchmark instance writes and reads its own set of three distinct files via sequential, backward, and random access. Concurrent processes never operate on the same files. We repeat the tests once for the on-node scratch file system and again for the off-node, shared, GPFS.

Figure 2 graphs the slowdown induced when concurrent, independent instances of I/O Bench run on a single node. The slowdown factors are far greater than those exhibited by the memory-intensive benchmarks, with the on-node numbers demonstrating super-linear slowdown. The off-node performance numbers, while not as egregious as their on-node counterparts, are nonetheless considerable. The erratic performance of the off-node measurements are likely an artifact of the varying demand placed on it by other applications concurrently executing on the system.

4 Mixing Jobs

Now that we have determined the ways in which resource sharing can degrade performance, we turn to investigate the extent to which this degradation can be mitigated by alternate job mixes. To gauge the performance effects of the benchmarks on each other, we repeat the experiments in Section 3, but utilize the unused processors in each experiment to concurrently execute other benchmarks instead of leaving those processors idle. We refer to the benchmark being tested as the *primary* benchmark and the one being executed by the spare processors as the *background* benchmark. To adjust for runtime

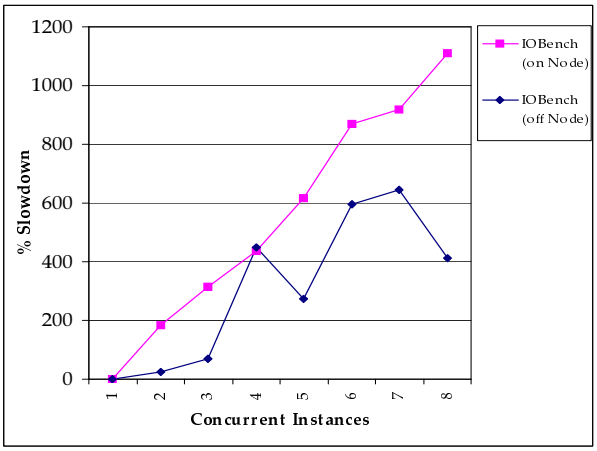

Fig. 2. Slowdown of I/O Bench as more instances run concurrently on a single node.

discrepancies between primary and background benchmarks, the processors executing the background benchmark repeat execution until the primary benchmark completes.

Figures 3 through 6 graph these results. In each graph, the line labeled "[BENCH] w/ idle" is the performance curve as depicted in Figures 1 and 2, meaning that there is no background benchmark and consequently, the unused processors were idle during the experiment. The other lines, labeled "[BENCH1] w/ [BENCH2]", indicate that instead of sitting idle, all unused processors were running the background benchmark BENCH2.

These graphs indicate that with only a single exception, utilizing unused processors to execute the other benchmarks has little to no effect on runtimes. These results clearly demonstrate that it is possible to mitigate resource-sharing slowdown by mixing memory, compute, and I/O intensive jobs.

The lone exception arises from combining the two memory-bound applications, STREAM and GUPS. Although GUPS has little effect on the performance of STREAM (Figure 3), the converse is untrue (Figure 4). This one-way interference is likely due to STREAM's heavy cache use and the relatively low rate of memory operations achieved by GUPS. STREAM increases the L2 and L3 miss rates of GUPS by around .2 each while the presence of GUPS does not affect STREAM's cache miss rates.

To confirm that these results are generalizable, we repeat part of these experiments using the NAS Parallel Benchmarks. Again, we use EP as the compute-intensive code and I/O Bench as the I/O-intensive code. Table 3 lists the percentage slowdown incurred by each primary benchmark, listed on the vertical axis, when all the remaining processors on the node concurrently execute the background benchmarks listed on the horizontal axis.

The first item to note is that EP and I/O Bench are symbiotic with all of the NAS benchmarks. The degradation imposed by these benchmarks both to and by the others is negligible and appears to be within the margin of measurement error.

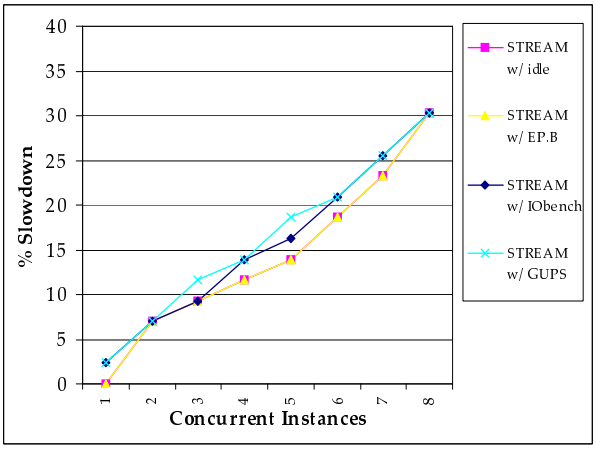

Fig. 3. STREAM

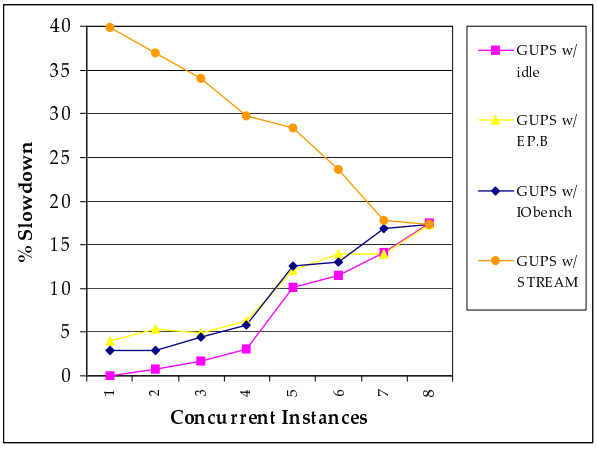

Fig. 4. GUPS

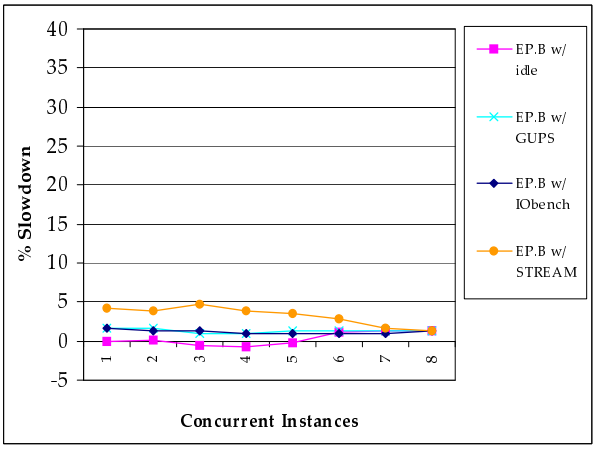

Fig. 5. EP.B

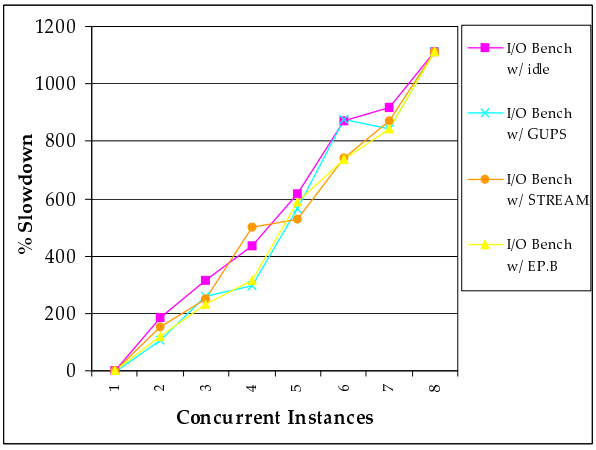

Fig. 6. I/O Bench

	Background Benchmark								
	BT	MG	FT	SP	LU	CG	IS	EP	I/O
BT	**12**	21	20	16	17	12	12	1	5
MG	11	**48**	25	25	25	11	11	1	4
FT	6	31	**30**	15	18	15	12	1	1
SP	21	48	36	**33**	31	23	19	2	5
LU	18	69	41	38	**41**	24	28	1	2
CG	26	82	64	42	55	**54**	36	3	7
IS	14	88	50	39	50	32	**58**	1	3
EP	2	4	4	4	4	3	2	**1**	2
I/O	-6	-2	2	-2	-6	-6	-2	-2	**1108**

Table 3. Percent slowdown of row application when all other processors on node execute column application.

Secondly, the slowdown imposed by each benchmark on itself tends to be among the highest observed. This implies that opportunities for symbiotic combinations may be forthcoming in large enough application sets.

As we observed in Figure 4, the interactions between the NAS benchmarks can sometimes be one-sided when one benchmark makes heavier use of shared resources than another. The most flagrant example is MG, which causes the most performance degradation both to itself and to others.

Degenerate cases aside, ample opportunities exist for symbiotic sharing, even among those applications we consider memory-intensive. Pairing CG and IS, for example, would be substantially beneficial to both. BT appears to be another possible candidate.

5 Symbiotic Space-Sharing and Parallel Codes

In the previous two sections we have shown that resource-sharing among concurrent jobs can cause performance degradation and that this degradation can be effectively mitigated by symbiotic job mixes. This is sufficient motivation to begin sharing single nodes among two or more small jobs in a more intelligent way. However, the question still remains as to whether or not symbiotic job scheduling can help speed larger, parallel, multi-node applications. This section aims to address this question.

Generally, parallel codes employ every processor on each node. The scheduler's motivation to use fewer nodes is to minimize the occurrence of slower, inter-node communications and therefore, ostensibly reap performance benefits. The results presented in the previous two sections should give us pause as to whether this is a good scheduling strategy. Can the processor performance benefits of symbiotic space-sharing outweigh the penalty of additional inter-node communications?

To answer this question we again use the NAS Parallel Benchmarks, only this time, instead of using multiple single-processor runs, we employ a single 16-processor run for each benchmark. We execute each parallel benchmark first on 16 processors spread

evenly across two 8-way nodes and then again on 16 processors spread evenly across four 8-way nodes, effectively utilizing only four processors per node.

To model the increased complexity of parallel I/O, we replace I/O Bench, which is a serial application, with BTIO [30], NPB's parallel I/O benchmark. BTIO is the same as the BT benchmark, but with frequent checkpointing to disk. There are several flavors of I/O that BTIO can utilize. We conduct our experiments using the following three:

MPI IO FULL - The full MPI-2 I/O implementation uses collective buffering, meaning that data scattered in memory among the processors is collected on a subset of the participating processors and rearranged before being written to file.

MPI IO SIMPLE - The simple MPI-2 I/O implementation does not leverage collective buffering, meaning that many seek operations may be required to write the data file.

EP IO - Using **E**mbarrassingly **P**arallel I/O, every processor writes its own file and files are not combined to create a single file.

The results in Table 4 were derived using the MPI implementation of the NPB version 3.2 with problem class C. Because we run these benchmarks across multiple nodes, the I/O tests cannot utilize the on-node I/O, but rather only the system's GPFS. Speedup is calculated using the traditional definition T_2/T_4 where T_N is the runtime of the benchmark on N nodes.

Benchmark	Speedup
BT	1.13
MG	1.34
FT	1.27
LU	1.47
CG	1.55
IS	1.12
EP	1.00
BTIO EP	1.16
BTIO SIMPLE	4.97
BTIO FULL	1.16

Table 4. Speedup of 16-processor runs when executing across four nodes instead of two

The results reveal that speedup from reduced resource contention in this benchmark set not only outweighs communication overheads, but does so significantly and consistently.

DataStar's current interface does indeed allow a user to request that his or her job be spread across more nodes than necessary and therefore attain these performance benefits. However, because the system does not node-share, such a request would be both a detriment to overall system utilization and costly to the user who is charged

per node instead of per processor. Can a system-level, symbiotic space-sharing scheme help?

To find out, we re-run some of the 4-node tests, but allow two benchmarks to run on the nodes concurrently. For each result presented in Table 5, we execute two parallel benchmarks concurrently on four nodes with each benchmark using exactly half of each node.

Bench A	Bench B	Speedup A	Speedup B
CG	IS	1.18	1.17
	BT	1.05	1.04
	EP	1.36	1.03
	BTIO(E)	1.38	1.07
	BTIO(S)	.55	1.03
	BTIO(F)	1.36	1.12
IS	BT	1.04	1.03
	EP	1.07	1.03
	BTIO(E)	1.11	1.07
	BTIO(S)	1.00	2.41
	BTIO(F)	1.13	1.13

Table 5. Speedup attained when parallel benchmarks share four nodes instead of running separately on two each.

These results show that speedup can be maintained even while no processors are idle. Speedup can be induced both by mixing categories of benchmarks and even by mixing some memory-bound codes. For the first time however, we observe some cross-category slowdown. CG and the BTIO benchmark with simple IO both slow considerably when paired. Nevertheless, these results demonstrate that executing parallel codes in symbiotic combination can indeed yield significant performance benefits. The average speedup increase is 15%, showing that for this set of benchmarks, the benefits of reduced resource sharing outweigh the increased cost of inter-node communications.

6 Towards a Symbiotic Scheduler

In the previous three sections, we have shown that symbiotic space-sharing can improve system throughput by reducing runtime inefficiencies while maintaining high system utilization. The most important question remaining is how to build schedulers that can leverage these concepts.

6.1 Identifying Symbiosis

The effectiveness of any symbiotic space-sharing scheduler is naturally contingent upon the level of symbiosis the scheduler can identify in a given job stream. In this section,

we discuss some preliminary approaches for uncovering symbiotic space-sharing opportunities under various assumptions.

In the most restrictive input scenario, the scheduler has no history of the execution characteristics of jobs in the stream. In such circumstances, users could be asked to submit the application's bottleneck, if any, to the scheduler. It is not unreasonable to assume that a user might know that a certain application is I/O or compute intensive. These are the two categories in which we are most interested since they afford us the most likely opportunity for symbiotic job mixing. The scheduler would then pair I/O and compute intensive jobs to execute with memory-intensive ones. As we will see in Section 6.2, even this naive approach can reap significant benefits.

If a scheduler, however, were able to recognize and maintain statistics regarding jobs that commonly recur in the stream, then other techniques would be possible. Workload traces have revealed that users tend to frequently resubmit similar or even identical jobs [9, 10], a phenomenon that automated runtime predictors have leveraged in the past [11, 21]. A symbiotic scheduler may utilize these same techniques to identify applications and associated resource bottlenecks. A user-supplied job category may be a good starting point, but the scheduler could improve on a strategy of random, cross-category pairing.

The most straight-forward approach is experimentation. The scheduler can be configured to space-share randomly selected cross-category pairs and learn the best combinations. *Better* combinations would be identified by metrics such as memory operations per second or floating point operations per second as reported by commonly available lightweight hardware counters. While sampling, the scheduler would exhibit a configurable bias towards choosing combinations known to be more efficient. This approach has been shown effective in multithreading scenarios [22, 23].

A yet more intricate approach may be to deploy those hardware counters to collect statistics on applications as they run alone. The scheduler may subsequently use the results to predict optimal combinations, thereby decreasing its learning overhead. Figure 7 exemplifies one possible predictive strategy.

Figure 7 graphs the memory operations per second achieved by the single-processor NAS benchmarks while running alone versus the percentage slowdown incurred by each when four or eight concurrent instances of it run on a single node. For the full node runs, we can see a strong correlation between these two parameters. Among the NAS benchmarks, those able to perform memory operations at a faster rate are less likely to cause themselves slowdown. Since the slowdowns incurred by the half-node runs of all benchmarks are comparable, it is likely that applications with lower memory operations per second will benefit more from symbiotic scheduling. In this approach, the scheduler might increase utility by preferring to space-share applications that achieve a lower rate of memory operations per second.

Aided by the proper hardware counters, a scheduler could ideally discover symbiotic combinations relatively quickly. Given that, there is still a need for effective scheduling heuristics that can exploit these findings. We should keep in mind however that an optimal symbiotic scheduler is not a necessary first step. The results presented hitherto suggest that much benefit would be achievable even by a naive implementation.

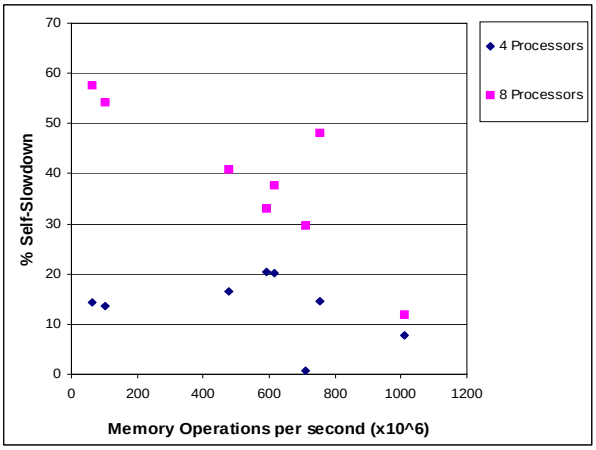

Fig. 7. Slowdown of NPB benchmarks as a function of memops/s.

6.2 Prototype Symbiotic Scheduler

To test whether or not symbiotic space-sharing can indeed improve system throughput, we implemented a rudimentary symbiotic scheduler to compete against DataStar's production counterpart.

The scheduler was deployed on DataStar and given an ordered stream of one hundred randomly selected 4 and 16-processor jobs to execute using four nodes. We refer to these job sizes simply as *small* and *large*. The jobs in the stream consisted of I/O Bench and variations of the NAS Parallel benchmarks from the following set: {EP.B.4, BT.B.4, MG.B.4, FT.B.4, DT.B.4, SP.B.4, LU.B.4, CG.B.4, IS.B.4, CG.C.16, IS.C.16, EP.C.16, BTIO_FULL.C.16}[2]. The job stream was generated by iteratively enqueueing jobs selected by weighted probability; small jobs were favored over large jobs in a 4:3 ratio and memory-intensive jobs were favored over compute and I/O intensive jobs in a 2:1:1 ratio. Each job is submitted with a synthetically generated expected runtime that is used by the scheduler for backfilling and is generally within 20% of the job's actual runtime.

The symbiotic scheduler mimics DataStar scheduling objectives by favoring large jobs and backfilling small ones whenever possible. To constrain backfilling opportunities, at most twelve jobs occupy the queue at any given time.

The algorithm employed by the symbiotic scheduler is the most simplistic of those discussed above. The scheduler partitions each node evenly into *top* and *bottom* halves. It then executes jobs designated as memory-intensive on the top half and all others on the bottom half. The symbiotic scheduler spreads large jobs across all four nodes while the DataStar scheduler executes each on only two.

DataStar's makespan for the first eighty seven jobs was 5355s while the symbiotic scheduler completed the same jobs in 4451s, a speedup of 1.20. We ignore the final

[2] For small jobs, each of the 4 processors actually performs a full serial run of the benchmark at class B with collective communication at the beginning and end.

thirteen jobs because the eighty seventh job completed was the final memory-intensive job in the stream. The symbiotic scheduler's memory half was thereafter starved while the non-memory intensive half executed the final thirteen jobs, an artifact of the testing procedure.

7 Related Work

Many previous investigations of multi-resource aware job scheduling have been conducted, though none under assumptions applicable to today's scientific supercomputing installations. Our approach revisits the issue by starting with a modern production policy on a large MPP machine and relaxing some procedures to achieve higher performance and utilization. We assume rigid job sizes, FCFS-type queued space-sharing, and run-to-completion scheduling with no preemption.

We characterize previous related work into the following non-exclusive categories:

7.1 Multithreading

Symbiotic job scheduling was originally proposed for machines utilizing Simultaneous Multithreading[22, 23], later known as *Hyperthreading*, and was subsequently refined by McGregor et. al. [18]. Such examples are concerned with intimate, cycle-by-cycle resource sharing of multithreaded processors where sharing and contention involve functional units on the processor. Contrastingly, this work focuses on space-sharing contention for off-chip resources by multiple processors.

7.2 Paging

Some studies have sought to schedule job combinations that may limit the amount of paging induced by the workload. In 1994, Peris modelled the cost of paging behavior in parallel applications when working sets would not fit into local memory [20]. Batat and Feitelson suggest limiting the multiprogramming level of gang schedules in order to ensure that job combinations do not exceed a total memory limit [7]. Suh and Rudolf have proposed that if such a limit must be breached, then previously obtained application profile information can inform the scheduler of the best way to do so [25].

Though ensuring a job's ability to fit into memory is encompassed by this work, it is not the sole focus. We address contention for all resources on each node including caches, memory bus bandwidth, and local I/O in addition to global resources shared among multiple nodes. We also study the effects of allocating a job's processes across multiple nodes in order to compare slowdowns from resource contention and inter-node communications.

7.3 Time-sharing

Application-aware job scheduling for time-sharing scenarios has also been studied. Many have proposed *affinity* techniques that mitigate cache perturbations by avoiding process migrations [24, 28, 27]. Such considerations are unique to time-sharing.

Wiseman and Feitelson have suggested that I/O and compute-intensive jobs can be symbiotically coallocated on the same processor set in a gang scheduled environment [29]. The focus of that work is on a relaxation of gang scheduling that allows two complementary jobs to cooperate via timely per-processor context switching. In contrast to this effort, our work targets resource sharing and contention in pure space-shared systems.

7.4 SMP Memory Bus Contention

It is well known that contention on the memory bus of an SMP is a scaling bottleneck. Several studies have therefore investigated the possibility of relieving pressure on this bottleneck through appropriate job mixes. Liedtke introduced the topic in 2000 [15]. Both Antonopoulos [5] and Koukis [13] have built upon his work by proposing techniques for scheduling jobs on SMP nodes in a manner cognizant of memory bus contention.

These studies are similar to ours in spirit, but target different environments. Koukis, for instance, targets serial applications which are time-shared on a cluster of dual-way SMP servers running Linux. The possibility of parallel applications is addressed but not evaluated. Contrastingly, we are concerned with space-sharing parallel scientific applications in production supercomputing environments under the assumptions detailed at the start of this section. We also study the effects of the cache hierarchy and I/O.

7.5 Other Related Work

Some previous studies of multiple-resource allocation have also been conducted. Parsons and Sevcik investigated the coordinated allocation or processors and memory [19]; subsequently, Leinberger et. al generalized the problem to *k-resource scheduling* where the idea is to choose optimal job working sets when multiple resource requirements exist [14]. Unlike our study, this work assumes independently allocatable resources and well defined requirements for each job. On our target architecture, a predetermined bundle of resources is provided to a job along with each processor.

It has been observed that spacing I/O-intensive jobs in time on a parallel file system improves performance [17]. Our emphasis is primarily to spread these in space, and also to identify specific symbiotic partners for such jobs.

Mache and Garg have focused on finding a spatial layout for concurrent jobs in a parallel space-shared machine to minimize communication and maximize access to I/O nodes for I/O-intensive jobs [16]. We address a related but different problem in considering not only the physical layout but also the sets of jobs contending for resources.

Also described has been an approach for deriving beneficial symbiosis (i.e. commensalism), wherein one version of a program, executing concurrently with the main program, helps the main program resolve control-flow for instruction fetching [26]. Alternatively, we search the existing job-stream for sets to co-schedule that interfere as little as possible with each other. We expect the existence of commensal job combinations in realistic production environments to be unlikely.

8 Conclusions and Future Work

In this work, we have introduced symbiotic space-sharing as a promising technique for improving the performance efficiency of large-scale parallel machines. We have shown that a wide range of benchmarks commonly suffer between 10-60% slowdown due to memory resource contention and up to several orders of magnitude for I/O. We have shown that this effect can be mitigated by deploying alternate job mixes and have extended these results to parallel codes, demonstrating that node-sharing among parallel applications can increase throughput by increasing performance while maintaining high system utilization levels. We synthesized these findings by exhibiting a prototype scheduler that improves throughput by 20%.

Our results are derived from DataStar, a production machine at the San Diego Supercomputer Center, and the NAS Parallel Benchmarks, a widely used benchmark suite designed with the express purpose of evaluating the performance of parallel Supercomputers.

Through this work, we have explored the opportunity space for and confirmed the viability of symbiotic space-sharing. Future work may proceed in the following directions:

The confirmed promise of symbiotic space-sharing warrants the effort to conduct a study on its applicability to real-world production workloads. The effectiveness of our techniques remains to be seen for highly parallel, resource-intensive, scientific applications. Further, a study characterizing the job mixes in today's production queues would help us understand more extensively the opportunity for symbiotic job mixes and the heuristics that could exploit them.

The relationship between hardware counter statistics and symbiotic space-sharing should be further explored. Such efforts could help create automated algorithms to identify the limiting resource of applications. A more advanced result might be to use such counters to automatically identify symbiosis, even among applications bound by the same resource.

Research on production workloads and prediction of job interactions can facilitate the development of symbiotic scheduling heuristics. Particularly interesting would be a framework for evaluating tradeoffs between system throughput and fairness in queue times or between other policy objectives.

We are currently also extending this feasibility study onto grid schedulers in an attempt to understand the degree to which a grid-wide scheduler can improve the efficiency of its resource pool by scheduling symbiotic job combinations at each site. Through this approach, we also hope to study the degree to which a scheduler can increase throughput by lessening site load on resources such as a parallel I/O file system.

9 Acknowledgements

This work was supported in part by the DOE Office of Science through the award entitled HPCS Execution Time Evaluation, and by the SciDAC award entitled High-End Computer System Performance: Science and Engineering. This work was also supported in part by NSF NGS Award #0406312 entitled Performance Measurement & Modeling of Deep Hierarchy Systems.

References

1. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.
2. http://www.cs.virginia.edu/stream/.
3. http://www.npaci.edu/DataStar/guide/home.html.
4. http://www.sdsc.edu/pmac/Benchmark/iobench.
5. C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. *icpp*, 00:547, 2003.
6. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
7. A. Batat and D. G. Feitelson. Gang Scheduling with Memory Considerations. In *14th Intl. Parallel Distributed Processing Symp.*, pages 109–114, 2000.
8. J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical Report ICL-UT-05-01, ICL, 2005.
9. A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, 1999.
10. Feitelson and Nitzberg. Job Characteristics of a Production parallel scientific workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 337–360. Springer, 1995.
11. R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 58–77, London, UK, 1997. Springer-Verlag.
12. S. Kannan, P. Mayes, M. Roberts, D. Brelsford, and J. Skovira. *Workload Management with LoadLeveler*. IBM, November 2001.
13. E. Koukis and N. Koziris. Memory Bandwidth Aware Scheduling for SMP Cluster Nodes. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 187–196, Washington, DC, USA, 2005. IEEE Computer Society.
14. W. Leinberger, G. Karypis, and V. Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 47, New York, NY, USA, 1999. ACM Press.
15. J. Liedtke, M. Volp, and K. Elphinstone. Preliminary thoughts on memory-bus scheduling. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210, New York, NY, USA, 2000. ACM Press.
16. J. Mache, V. Lo, and S. Garg. Job Scheduling that Minimizes Network Contention due to both Communication and I/O. In *14th International Parallel and Distributed Processing Symposium*, page 457, Washington, DC, USA, 2000. IEEE Computer Society.
17. J. Mache, V. Lo, M. Livingston, and S. Garg. The impact of spatial layout of jobs on parallel I/O performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 45–56, New York, NY, USA, 1999. ACM Press.
18. R. L. McGregor, C. Antonopoulos, and D. Nikolopoulos. Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005. IEEE Computer Society Press.
19. E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 57–67, New York, NY, USA, 1996. ACM Press.

20. V. G. J. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 5–18, New York, NY, USA, 1994. ACM Press.
21. W. Smith, I. T. Foster, and V. E. Taylor. Predicting Application Run Times Using Historical Information. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, London, UK, 1998. Springer-Verlag.
22. A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, November 2000.
23. A. Snavely, D. Tullsen, and G. Voelker. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the ACM 2002 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS. 2002)*, pages 66–76, Marina Del Rey, June 2002.
24. M. Squillante and E. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systemse*, 4(2):131–143, February 1993.
25. G. E. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 116–132, London, UK, 2001. Springer-Verlag.
26. K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.
27. J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139, February 1995.
28. R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 26–40, Pacific Grove, CA, October 1991.
29. Y. Wiseman and D. Feitelson. Paired Gang Scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 14, pages 581–592, 2003.
30. P. Wong and R. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Ames Research Center, Moffett Field, CA 94035-1000, January 2003. NAS Technical Report NAS-03-002.