Weighted locality-sensitive scheduling for mitigating noise on multi-core clusters

Vivek Kale*, Abhinav Bhatele† and William D. Gropp*
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
E-mail: vivek@illinois.edu, bhatele@llnl.gov, wgropp@illinois.edu

Abstract

Recent studies have shown that operating system (OS) interference, popularly called OS noise can be a significant problem as we scale to a large number of processors. One solution for mitigating noise is to turn off certain OS services on the machine. However, this is typically infeasible because full-scale OS services may be required for some applications. Furthermore, it is not a choice that an end user can make. Thus, we need an application-level solution. Building upon previous work that demonstrated the utility of within-node light-weight load balancing, we discuss the technique of weighted micro-scheduling and provide insights based on experimentation for two different machines with very different noise signatures. Through careful enumeration of the search space of scheduler parameters, we allow our weighted micro-scheduler to be dynamic, adaptive and tunable for a specific application running on a specific architecture. By doing this, we show how we can enable running scientific applications efficiently on a very large number of processors, even in the presence of noise.

1. Introduction

The emergence of multi-core clusters has brought great promise to attaining better performance, but has also introduced several challenges. One challenge is getting good performance from a multi-core node without requiring a large amount of effort by the application programmer. This is a top priority for running applications on clusters with nodes consisting of many cores.

There is also the broader issue of performance *consistency* that arises for large-scale machines, which affects overall performance. Recent studies have shown that inconsistent performance on one node, most notably attributed to OS interference or noise during computation, is amplified as we scale to a larger number of nodes [1]–[3]. In particular, for applications that involve several successive iterative time steps with collective communication in between, localized performance perturbations seen on one multi-core node can be amplified across a large-scale cluster. This is because during every time step, some processor is likely to experience a noise event delaying all other processors at the collective operation. Such noise amplification hinders scalability of an application to hundreds of thousands of nodes.

Petrini *et al.* [1] raised the importance of the issue of system noise through an in-depth study of the ASCI Q cluster. Their study shows the impact of localized perturbations through several performance tests that bring to light the impact of cascading effects of noise amplification that accumulate over several iterations of an application. There have been follow-up studies on the impact of system noise on several well-optimized applications, indicating the tension and separation between OS services and application-specific tuning done for an architecture [4]. Some research projects such as FastOS have tried to develop light-weight operating systems, which minimize localized system interference on a compute node. Other solutions involve co-scheduling all OS daemons simultaneously on all nodes, in an effort to obtain more predictable performance. Finally, a recent study done on the Cray XT5 machine at the Oak Ridge National Laboratory shows that removing work from the most noisy core of a node gives more predictable performance and allows for achieving better application scalability [5].

Stripping away system services to solve the noise problem can, in some cases, degrade application performance. Considering the case of a regular mesh computation, operating system activity such as memory management may actually provide performance benefits which the application programmer may not otherwise be able to tune for. However, if an OS service is turned off, the programmer may then have to manually implement and tune the corresponding functionality. The programmer's manual optimization may succeed for one particular machine, but if a new platform (specifically, a new OS) is released, the programmer would have to rewrite the code so that it is tuned for this new operating system.

We suggest an alternative solution. Our solution involves no modifications to the OS or to the underlying runtime. We argue that a "higher-level" solution is needed to deal with the problem of system-induced load imbalance. Our specific strategy is to identify computation in the application that can be broken down into fine-grained tasks, and then dynamically schedule these fine-grained tasks through the use of a queue shared across cores of a node. Yet, as soon as we argue this, there are several obstacles that we must address before we can claim this to be an acceptable solution

that programmers can make use of. First, dynamic scheduling can destroy properties of locality that may have been well-established in the original data parallel, i.e. statically scheduled algorithm. Second, the overhead of pulling a task from the queue can also cause dynamic scheduling to be worse than static scheduling. The overhead often comes from having to lock the task queue, to pull a task from it and update the queue appropriately.

The cost of coherence cache misses incurred by a load balancing technique, due to the arbitrary movement of data from one core to another, is also of significant concern. The cost of a coherence miss varies significantly from machine to machine, and is often dependent on the memory hierarchy of the machine. A single universal scheduler is often not adequate for many scientific applications that are being used today. To employ an effective scheduling strategy requires tuning for a given architecture and operating system, in a search space that has various discontinuities and is not completely well-defined. For these reasons, much classic literature on fundamental computations on dense matrices, such as LU or QR factorization, or structured grid computations, has avoided the dynamic scheduling of kernels. Conventional wisdom tells one to use static scheduling and domain decomposition techniques to get optimal performance for such codes.

However, a preceding study that we did shows that through careful tuning to keep the cost of dynamic scheduling low while trying to minimize small-scale load imbalances within a node, we can achieve more predictable performance within a multi-core processor; this ultimately allows for better scalability as we run on a larger number of nodes [6]. The amount of dynamic scheduling we allow is proportional to the duration of the characteristic system noise of a machine. To keep the costs of scheduling low, we use relatively simple information in the task data structures which identify the thread on which a task ran on in a previous time step of the application. We refer to this light-weight application-level load balancing as micro-scheduling.

As discussed in our previous work, we use a simple load balancer that uses dynamic scheduling to assign work to processing elements when they are ready, but this is taken one step further to make every effort to reduce overheads of the scheduler, in particular to reduce performance penalties due to coherence cache misses. Our initial work shows that the performance improvement of an application is relatively unnoticeable when running on a small number of nodes of a cluster, but becomes much more dramatic as we scale the application to a large number of nodes [6].

In this work, our key contribution is an augmentation of our reactive, queue-based micro-scheduling approach with a form of pro-active load-balancing[1]. Pro-active load balancers can assign work to each core in proportion to its availability w.r.t. OS services running on it. We refer to this augmented load balancing strategy as weighted micro-scheduling. We show the benefits, over our original solution, for aiding noise mitigation by:

1) discussing an implementation of this technique that is portable and efficient for regular computations.
2) validating our implementation's efficiency by comparing it against industry standard scheduling such as OpenMP guided scheduling.
3) showing how we can tune our scheduler using a weighted factoring approach [7] that assigns less work to slower cores and more work to faster ones.

1. Pro-active load balancing adjusts load before the application time step begins, while reactive load balancing adjusts load during the time step.

2. Architectures considered

The two architectures we consider are ORNL's Cray XT5 machine (Jaguar) and TACC's SUN constellation cluster (Ranger). An in-depth performance comparison of the two supercomputers is presented in [8]. Each node of Jaguar consists of twelve cores, 16 GB of memory and generates a peak flop rate of 124.8 Gflop/s. The nodes run a specific version of the SuSE Linux operating system that has been optimized by Cray. While Cray has not released the operating system software as open-source, they claim to have tuned the kernel to remove unnecessary OS services from compute nodes.

Each node of Ranger consists of sixteen cores, with a peak flop rate of 147.2 Gflop/s per node. The frequency of each core on Ranger is 2.3 GHz and allows for four floating point operations per clock period. All cores within a node share 32 GB of memory. In this case, the operating system used is the Linux kernel (release 2.2) from kernel.org and is open-source.

A characterization of system noise for the two systems, Jaguar and Ranger was done through the use of a sequential computation run for several thousand iterations on each core. This represents the fixed work quantum (FWQ) method of recording noise. The timings for the sequential computation for each iteration on each core of the machine were recorded. Figure 1 shows the minimum and maximum execution times for the work quantum on 100 nodes or 1200 cores of Jaguar. For most cores, the maximum time spent in execution is several times the best execution period (14–15 μs). We can also see the impact of two specific daemons of 100 μs and 200 μs durations respectively.

Figure 2 shows a similar plot of the minimum and maximum execution times for 100 nodes or 1600 cores of Ranger. In this case, the maximum execution times are random for different cores, although, the spread is denser in the 17–100 μs region. In order to get a better idea of the distribution, the execution times for the sequential computation for all cores across all iterations were placed into 5 μs bins and histograms were plotted for the two

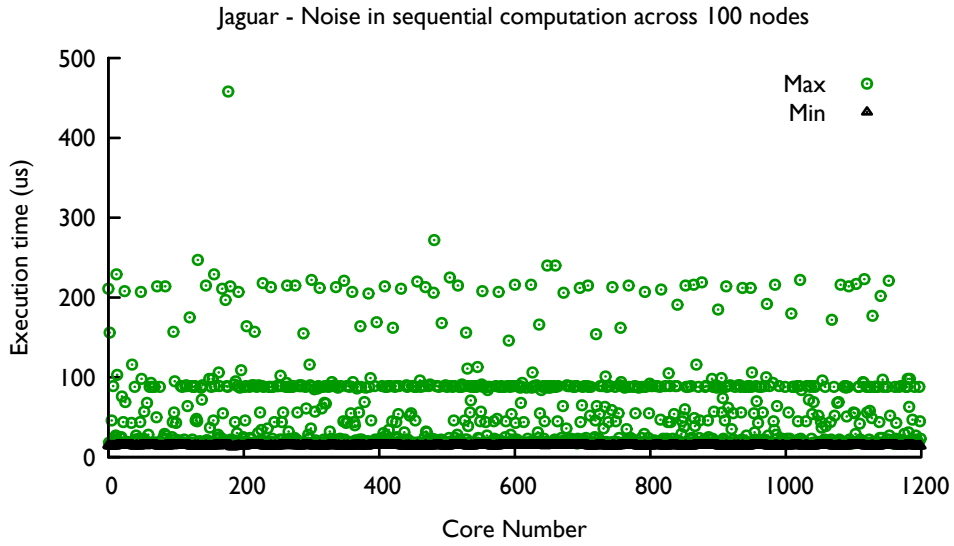

Figure 1. System noise plotted against all ranks for a 100 node run on Jaguar

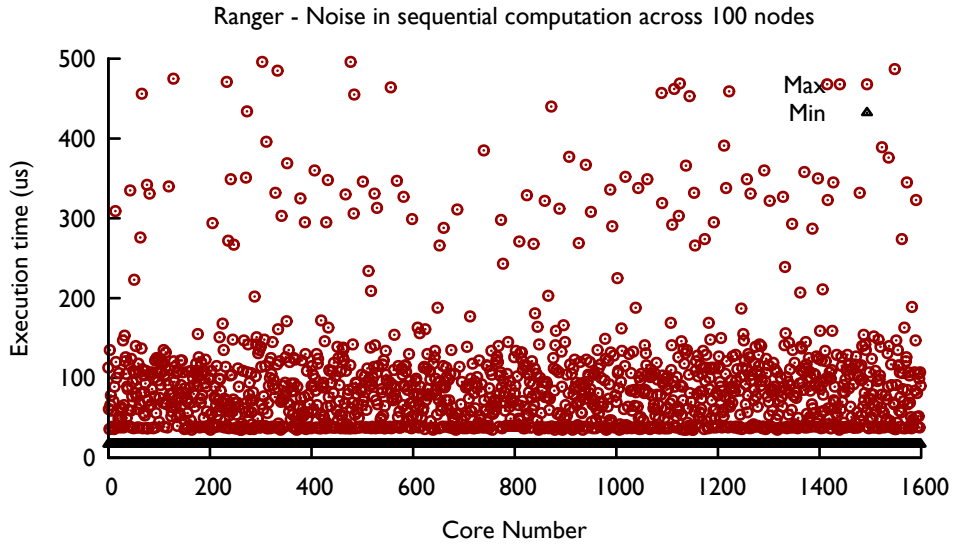

Figure 2. System noise plotted against all ranks for a 100 node run on Ranger

machines (Figure 3). The labels on the x-axis are bin-numbers, so a label "60" corresponds to a bin representing execution time between 300–305 μs (also note that the scale on the y-axis is logarithmic). For Jaguar (left), there are four distinct peaks, last three of which possibly correspond to specific daemons that have a high frequency. The Ranger plot (right) shows a spread from 15–200 μs and then a smaller distribution around 300 μs.

Given a completely noise-free machine and an application that has no load imbalance, static scheduling and proper domain decomposition techniques would be most effective. Owing to the load imbalance induced by the operating system, it is clear that some level of dynamic scheduling can be beneficial. Several previous studies (dating back more than 30 years) have shown dynamic scheduling to be beneficial for machines that have unpredictable behavior. Yet, the above characterization of the inherent noise on a system suggests that traditional dynamic scheduling techniques are not sufficient for applications whose performance depends heavily on the architecture.

Each characteristic of the noise pattern of a particular platform suggests a particular feature of a dynamic sched-

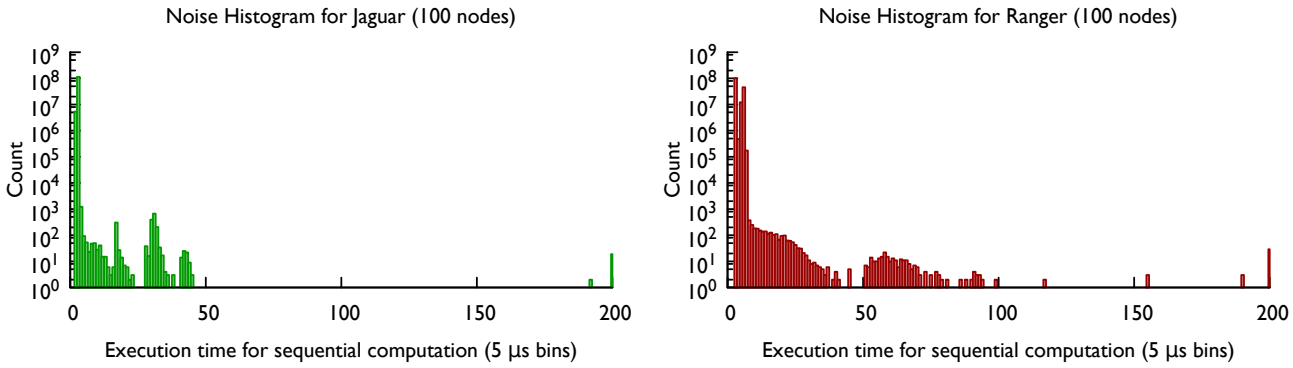

Figure 3. Histograms for the execution time of a sequential computation performed to record noise events on Jaguar and Ranger. The labels on the x-axis are bin-numbers.

uler. The above characterization confirms the benefits of the solution proposed in [6]. When a platform has several different noise events of different lengths, a dynamic scheduling strategy with an assortment of task granularities can be used. If there are only a few different noise events with relatively similar durations, we can tune the task granularity. If the frequency of a noise event is so high that there are multiple noise events within an iteration, we can tune the ratio of dynamic to static scheduling.

In order for a scheduling strategy to be effective, it is crucial to take the constant cost of obtaining a task from a queue into account. As described in [9], using multiple queues can reduce synchronization overheads. The number of queues used depends on the synchronization overhead on an architecture. The use of multiple queues is one of the key features of work-stealing in Cilk.

3. Load balancing techniques

Locality-sensitive scheduling aims to schedule work on cores that executed this work in a previous time step. This optimization aims to improve temporal locality, and has been shown to provide benefits in [6]. The strictness of locality constraints depends on the cost of a cache miss. For applications with many iterative steps, locality-sensitive scheduling can reduce the impact of cache misses.

Our basic scheduler [6] uses a queue to assign chunks of work to cores. Each queue is implemented as an array of pointers to data structures we refer to as *tasklets*. A *tasklet* is a fine-grained piece of computation which keeps track of its movement across cores of a node. A queue can consist of an assortment of tasklets of different lengths. These tasklets are dynamically scheduled across cores of a node, but are aware of their movement across cores of a node.

In this work, we improve our technique by augmenting the statically scheduled stage of the computation with weighted scheduling. Our previous work assumed that a noise event was equally likely to abduct any core of a node, and that a dynamic scheduler could, in theory, handle noise events of any duration and any frequency. Yet, short-duration, high frequency events are fine-grained enough that the application quantization of tasklets limits the ability to distribute the noise across cores evenly. Our previous work did not handle short-duration, high-frequency noise events specially.

We address the case when a core is continuously impacted by short-duration, high-frequency noise events. We observe that a core effectively becomes "slow" when there are many such OS daemons that are bound to that particular core and must be frequently time-sliced with the actual computation. Due to the many different system services that are running on that core, we observe that the core may have a higher chance of performance degradation during a time step. Finally, only a subset of cores tend to have several OS daemons running on them, while the remaining are relatively unoccupied with such system services [10]. If we know that some cores are always slower than others (all of the time), we can employ a strategy to offload some work from the slow cores, and move that work onto faster cores.

Putting both techniques together, we use a prescriptive load balancing technique in the first stage of the computation to mitigate system noise, followed by a reactive load balancing technique in the second stage of the computation. This strategy tries to reduce the amount of work done on those cores that are heavily occupied by the OS services.

Using weighted scheduling, we obtain performance gains when we are able to correctly (or nearly correctly) predict which processors likely remain slow throughout the duration of the application time step. This allows all processors on the node to start their work at the same time in the subsequent dynamic scheduling stage. If we know this information before an application time step begins, we can reduce the chance that the slow core(s) impacts the collective iteration time across all cores. This can be done by using a form of measurement-based load balancing as in Charm++ [11].

Weighted scheduling works well because it offloads work proportionally to the speed of a processor. Its advantage over dynamic scheduling arises from the fact that it involves no dequeue overheads for locking and unlocking the work queue. Weighted scheduling tries to enforce that the slow core's static section will not be the cause of time step slowdown. If it was the cause of the time step slowdown, the static section will essentially "protrude out". In this case, there is no way for the other cores to "steal" the work from the static section. However, weighted scheduling does also have its shortcomings. Noise events may not necessarily be restricted to one, or a subset, of the cores on a node. Thus, the predetermined weighted factoring will serve little purpose. In addition, weighted scheduling does not handle low-frequency, long-duration noise events. These low-frequency, long-duration noise events that can happen on any core are seen on both Jaguar and Ranger, and thus should not be ignored.

The set of cores that are slow varies for different platforms. Even for a particular platform, the set of cores that are slow can change over the course of days or weeks. To handle both of these issues, we run an initial loop with a square-root computation to gather the speeds of each core before the application begins. We also allow for adjustment of weights during runtime, to handle the case when the speeds of the cores change during execution of the program.

Also, for the dynamic stage, we make use of a more systematic auto-tuning methodology to find the best scheduler parameters. The tuned parameters include the average tasklet granularity and tasklet granularity distributions. Our automated tuning is done in the form of a shell script that runs before execution of the program. In the future, we hope to make our auto-tuning more sophisticated.

4. Results

To assess the effectiveness of the proposed techniques, we use a three-dimensional five-point stencil computation that is described in [6]. The computation and its domain decomposition for the MPI only (no pthreads) implementation is shown in Figure 4. A one-dimensional slab decomposition of the data array is done and a slab is assigned to each MPI task. In the hybrid MPI+pthreads implementation, each thread is assigned a portion of the slab as shown in Figure 5. Each compute thread corresponds to a core of a node of the cluster. Each MPI process corresponds to a node of a cluster. We make note that there are ways to optimize the process/thread aspect ratio, but we use the simplest one here, as we see that tuning in this search space does not give a large performance difference for our particular stencil code.[2]

2. This optimization is under a category of static auto-tuning strategies, and is out of the scope of this study (at least for what want to investigate for the time being). However, we do believe we can integrate such auto-tuning techniques into our scheduler tuning.

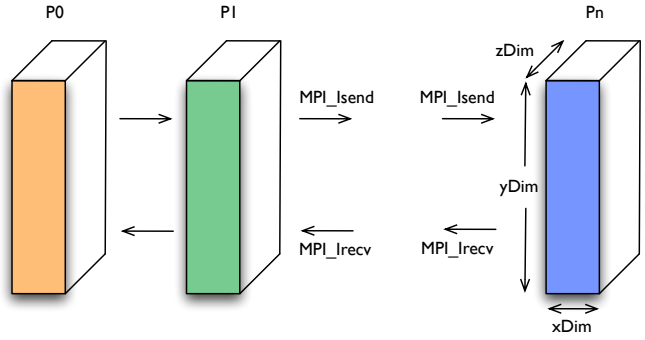

Figure 4. MPI domain decomposition used for the 3D Stencil code.

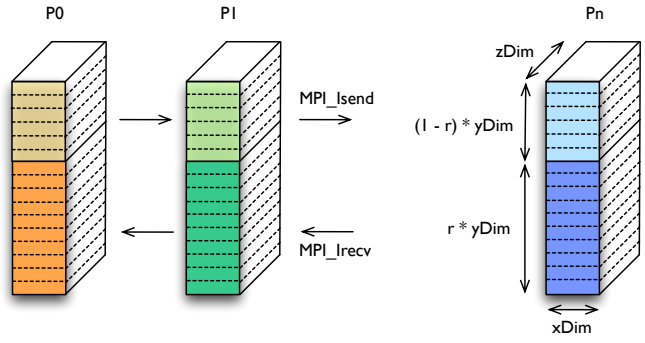

Figure 5. Hybrid MPI+pthreads domain decomposition with static and dynamic micro-scheduling used for the 3D Stencil code

Figure 6 shows the MPI+pthreads implementation that uses weighted scheduling in the first stage of the computation. Here, w denotes the weight of the work assigned to a particular thread. We run the 3D stencil computation for 1000 iterations, and use a problem size of $64 \times 32 \times 64$ for each core, regardless of the machine we test on. By ensuring that we consider a dense matrix with regular computation, we can more easily isolate the problem of noise for different machines. Below, we show how each of the two schedulers perform with respect to the baseline static scheduling. We also show a comparison to commercial schedulers, such as OpenMP guided scheduling [12] and the TBB affinity scheduler [13]. Through careful tuning of the parameters shown in Figure 6 in our experimentation, we can obtain significant performance benefits for the stencil application.

We now proceed to the discussion of the performance of weighted scheduling, micro-scheduling, and our combination of the two schedulers. Figures 7 and 8 show the performance of the stencil computation described above, using the various schedulers on Jaguar and Ranger respectively. On Jaguar, we ran the stencil code on up to 1024 nodes and Ranger, we ran on up to 512 nodes.

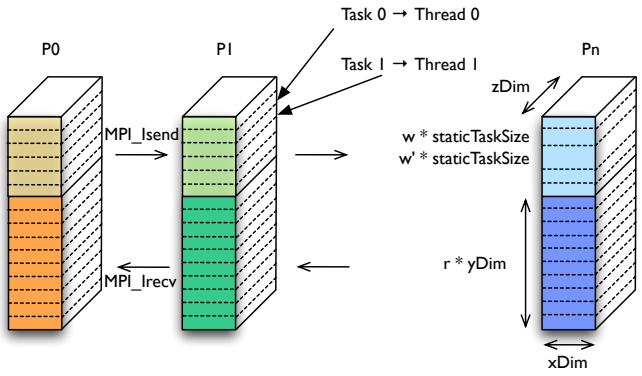

Figure 6. Hybrid MPI+pthreads domain decomposition used with weighted static and dynamic micro-scheduling for the 3D Stencil code. The dynamic ratio is denoted by r.

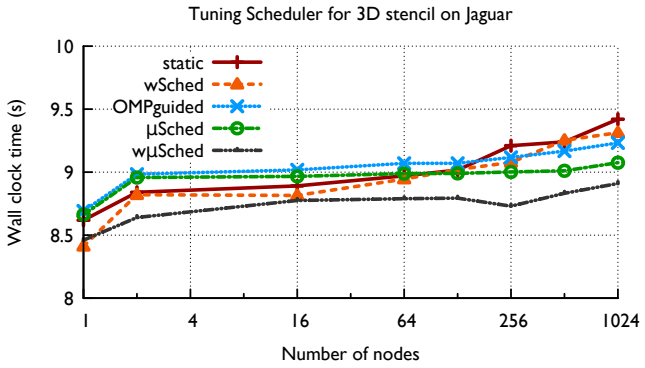

Figure 7. Performance of the stencil computation on Jaguar for various scheduling techniques.

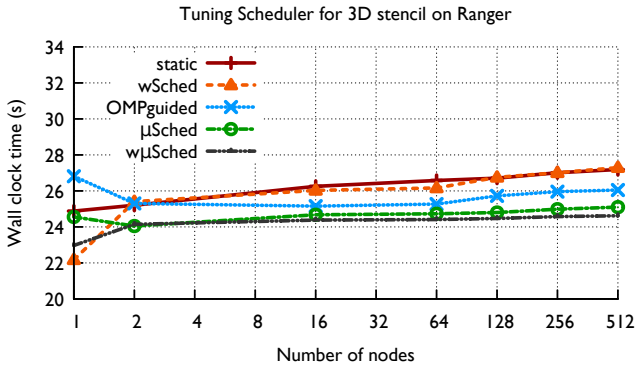

Figure 8. Performance of the stencil computation on Ranger for various scheduling techniques.

4.1. Weighted Scheduling

Running on 512 nodes of Jaguar, weighted scheduling gives a benefit of 6% over the baseline static scheduling (Figure 7). This is somewhat better than the performance gain that micro-scheduling gives us. Since OS noise typically affects a subset of the cores on a node [8], weighted scheduling is primarily beneficial at the beginning of the computation. By offloading work from the noisy cores of Jaguar, we have provided a solution that is better than micro-scheduling in the sense that it avoids dequeue overheads that the non-noisy cores would otherwise have to suffer.

Running on 512 nodes of Ranger, weighted scheduling gives almost no performance benefit over the baseline static scheduling (Figure 8). There is a smaller benefit of weighted scheduling on Ranger due to the fact that noise can occur on any core of a node, rather than being restricted to a subset of cores. Note that unlike micro-scheduling, weighted scheduling incurs no dequeue overheads. Weighted scheduling will only incur idle time due to measurements that may mispredict weights, or because of low-frequency noise events that affect only a few time steps. Because our scheduler is conservative, the performance loss of trying to use weighted scheduling on Ranger is very low.

4.2. Micro-scheduling

To assess the effectiveness of tuned micro-scheduling, we use an automated tuning to search for the best parameters for the dynamic scheduler on each architecture. Micro-scheduling provides for the most performance benefit on Ranger, with 12% performance improvement on 512 nodes. This is likely due to the fact that any core on a Ranger node can get perturbed by a noise event, not a specific subset of the cores. Because the noise interruptions occur over a range of short and long durations, rather than over a fixed duration, the task granularity cannot be determined ahead of time. By using variable-sized tasks, we are able to handle these variable-length noise events. It should be noted that the use of variable-sized tasks resembles guided scheduling, as implemented in OpenMP.

On Jaguar, the tuned micro-scheduling strategy is less effective. The reason is that the noise events happen on only some subsets of cores. These noise events are also probably of higher frequency. While using variable sized tasks can help address the issue of variable duration noise events, we believe that our strategy of using variable-sized tasklets does not help as much due to the fact that our implementation is likely not tuned carefully. With further tuning of the tasklet length distribution in the queue (effectively, finding the right "assortment" of tasklet lengths), micro-scheduling can provide for better performance on Jaguar. However, this will require more investigation, and we leave this as future work for now.

4.3. Weighted micro-scheduling

As the preceding sections show, noise that occurs sporadically on any cores of a node can be mitigated by micro-scheduling, and noise that is restricted to run on a subset of nodes is well-mitigated by weighted scheduling. However, in practice, production HPC clusters may not have a clear distinction between the case of noise being focused on a subset of the cores versus the case where noise occurs sporadically on any core of a node [10].

For example, noise events on a production HPC cluster may be more *likely* (rather than restricted) to occur on one core than the other. Thus, using either of the schedulers in isolation does not necessarily provide for significant performance gains. To what extent can we combine these two different schedulers to get the best of both worlds? We now discuss the results obtained when we combine the weighted scheduler and micro-scheduler.

On Jaguar, we observe that noise happens on some cores more than others. Specifically, cores 0, 6, and 9 are slowest. This is 3 out of the 12 cores that could potentially finish late and cause a slowdown. By using weighted micro-scheduling on 512 nodes of Jaguar, we get a performance improvement of 12% over the baseline static scheduling. This is significantly better than the performance we get with using the weighted scheduler (7%) or the micro-scheduler (9%). In this case, the knowledge that these cores are noisy allows us to reduce the time for each time step. In addition, dynamic scheduling handles low-frequency, small duration noise events occurring on any core.

On 512 nodes of Ranger, weighted micro-scheduling offers a performance improvement of 16.6%, which is a slight improvement over the 14.1% improvement we get with just micro-scheduling. All cores are almost equally noisy, though core 0 has slightly more noise than other cores. Weighted scheduling (offloading work from core 0) does help to mitigate high-frequency, short duration noise on core 0. However, the benefit of weighted scheduling is still not significant; the benefits obtained through micro-scheduling still dominate in the results for Ranger.

A key observation we make here is that while weighted scheduling is not very successful on Ranger, our combined weighted micro-scheduler also does not hinder performance, compared to the corresponding results for the weighted scheduler or micro-scheduler. Thus, our solution of combining weighted and micro-scheduling is portable; when we do not have "slow" cores on a node, the weighted scheduling portion of the combined weighted micro-scheduler does not induce unnecessary overhead. Further, our scheduler will be able to handle static and dynamic variations that are likely to arise in future architectures due to semiconductor process variation, cache error correction, etc.

5. Further experimentation

In this section, we present the impact of application parameters on the performance of our scheduling techniques. We focus on the memory accessed per iteration. In order to isolate performance efficiency of our scheduler from the efficiency of the MPI runtime, our experiments are performed on one node.

5.1. Impact of computation per time step

We vary the length of the time step to identify the impact of computational noise as we decrease the communication-to-computation ratio. Figures 9 and 10 show the performance for different time step lengths. We vary the length of the time step by changing the number of times the stencil computation is repeated before doing an update with neighboring processing elements (i.e. cores).

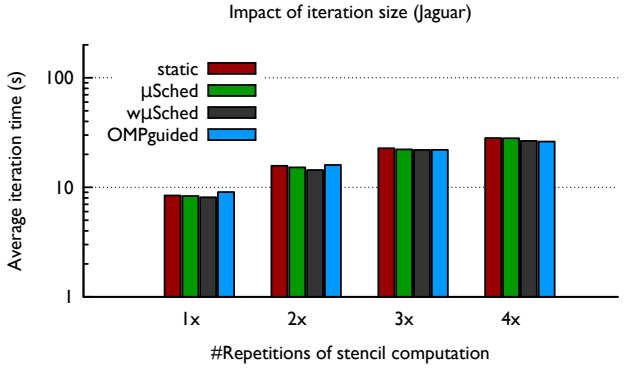

Figure 9. Impact of varying the amount of computation done in each time step on Jaguar for 3D Stencil with different load balancing strategies.

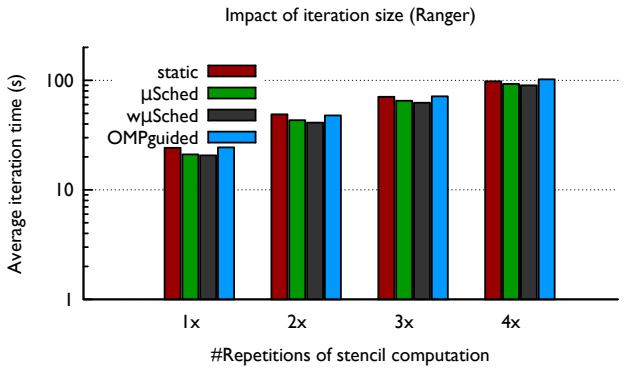

Figure 10. Impact of varying the amount of computation done in each time step on Ranger for 3D Stencil with different load balancing strategies.

As is seen in Figures 9 and 10, our strategies are competitive with and seem to perform better than the standard OpenMP guided scheduling. We expect to get better performance with further tuning of our scheduler's task granularity, and through further enhancement of the support for locality. In addition, we can make our technique more efficient by using a more precise weighted factoring scheme.

When considering the baseline static scheduling, an application with larger time steps seems to suffer less from system noise. On both machines, the performance gains using weighted micro-scheduling do not vary significantly with more computation within the time step. On Jaguar, we observe that the amount of computation within an application time step does not make a significant impact on how much benefit weighted micro-scheduling offers. A likely reason that the amount of computation in the application time step may make a difference for Jaguar is that high-frequency short-duration noise bound to particular cores has a relatively larger presence on this machine. Such noise will persistently affect a core throughout the duration of the time step. Thus the overall impact of noise on load imbalance increases linearly with the amount of computation done per time step.

5.2. Impact of memory accessed per time step

Figures 11 and 12 illustrate the impact of varying the problem size in this stencil computation on each of the machines. In the figures, the third cluster of bars from the left indicates the baseline problem size. As is seen in these figures, our strategies are again competitive with and seem to perform better than the standard OpenMP guided scheduling. By varying the problem size, we are able to test how time for memory access impacts our strategy. Results on both Jaguar and Ranger, with a very small problem size suggest that system noise is clearly a factor in performance. The lack of benefit for larger problem sizes is likely due to the fact that performance is already impacted by memory bandwidth limitation.

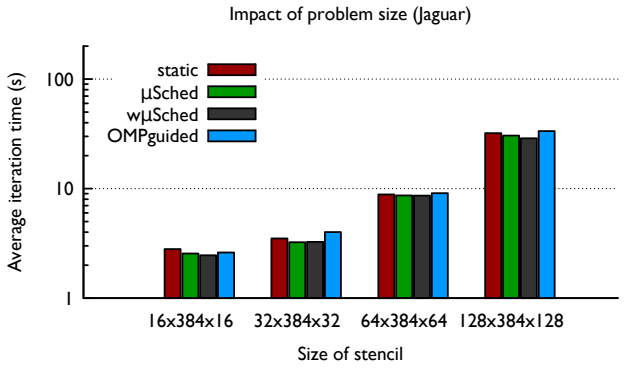

Figure 11. Impact of problem size on Jaguar for 3D Stencil with different load balancing strategies.

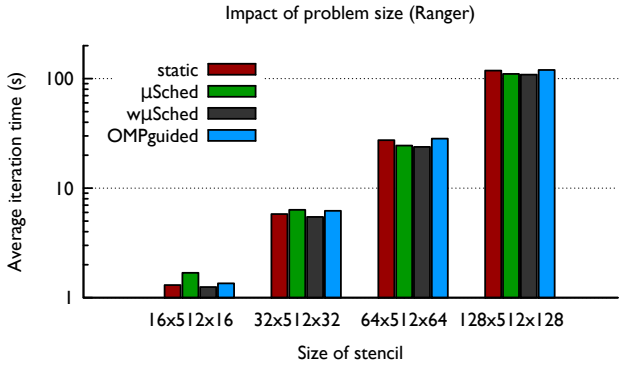

Figure 12. Impact of problem size on Ranger for 3D Stencil with different load balancing strategies.

6. Discussion

This work builds on previous work that used our scheduling techniques for hybrid MPI+pthread programs. We aim to build a more acceptable strategy for use at the application level. There are two methods behind each of these techniques: the first is auto-tuning and the second is measurements of performance taken from previous application time steps.

Offline auto-tuning happens for the dynamic phase, while online auto-tuning is used for the static region. Auto-tuning is important because system noise typically does not change during the execution of the program. For a long running application, the subset of processing elements that are slower may change. Furthermore, if there is a large noise event that spans several iterations, the scheduling techniques will serve no benefit. In this case, we will need a measurement-based load balancing technique such as in Charm++ [11] that adaptively adjusts to situations where cores are constantly slowed down by operating system events, even across time steps. We must move work away from those "slow" processing elements.

Note that there is no pro-active load measurement involved in our solution of weighted scheduling. An intelligent load balancer like that in Charm++ uses sophisticated techniques to implement load balancing strategies by collecting load balancing information from previous iterations [14]–[16]. The nature of load imbalance we have observed is not very predictable. However, if we are able to find that there is a pattern in the noise events, an intelligent measurement-based technique that uses periodic load balancing may be advantageous. Further, our technique can be used in conjunction with Charm++'s measurement-based load balancers.

Finally, we make a note that performance tuning a scheduler for a given operating system is a difficult task. If

applications are sensitive to the operating system that they are running on, the argument for portable application code is further justified.

The rapid pace of innovation of computer architectures is clearly evident, and can change over just a year's notice. This requires one to continually re-tune application codes for these new architectures. Unlike the pace of innovation for architectures, an operating system can change underneath within weeks. Furthermore, operating systems are programmed by humans. This further adds complications because humans make mistakes, thereby causing performance (or correctness) bugs in kernel software.

7. Related work

There are several studies that provide an in-depth analysis of noise and its impact on large-scale systems [1]–[3], [17], [18]. Beckman *et al.* discuss a benchmark for quantification of noise referred to as the selfish benchmark [19]. This benchmark has enabled a more accurate and proper study of noise in several follow-up studies, and has allowed one to quantify noise on a system in a standardized fashion. A study of the impact of noise for MPI applications was done by Hoefler *et al.* [3]. This showed how noise can have a large impact for a particularly large number of MPI processes. This assessment of performance loss is done for several key scientific applications, and many different experiments are carried out to understand changes in performance as the amount of system noise is increased.

Such studies have provided insight on how to mitigate noise, and several studies show its usage. The study by Petrini *et al.* of OS system services on ASCI Q investigates how to mitigate noise [1]. Noise mitigation is achieved through system service suppression i.e. the sources of system noise (in the form of OS daemons) are identified, and then each non-critical service is stripped away from the machine. Stripping away OS services was key to enabling better performance of ASCI Q. They also suggest using co-scheduling to make existing noise more coordinated. Furthermore, the study discusses several lessons learned from their studies, particularly the impact that noise mitigation solutions have on different classes of applications. A paper on NAMD argues that effects of noise in the communication sub-system can be mitigated by data-driven execution [20]. Not all applications can use this technique though. For some applications, overlap of communication and computation is not possible due to inherent strict dependencies across application time steps.

The above solutions, particularly those discussed in Petrini *et al.*, are lower-level and specific to the platform used to run the application. They not portable and general to enable applicability to a larger class of machines and a larger set of applications. As discussed in [6] and [9], a higher-level solution is necessary to take advantage of the compute power of emerging clusters of multi-cores. In particular, the application programmer should have control over how the application should be implemented, ensuring that noise has minimal impact. In this work, we particularly suggest that measurement-based load balancing can be used for system noise mitigation, and use it to improve a dynamic scheduling strategy implemented in [6]. Note that in order for a scheduling strategy to be effective, it is crucial to take the constant cost of obtaining a task from a queue into account. As described in [9], using multiple queues can reduce synchronization overheads. The number of queues used depends on the synchronization overhead on an architecture.

8. Conclusion

This work extends upon previous work that used hybrid static+dynamic scheduling to improve performance of high-performance scientific codes. The strategy allows processing elements to start work whenever they are ready. Since each architecture has different parameters of dequeue overhead and cache miss penalty, etc., it is important to establish a more general, portable solution that can tune itself, as well as adapt to the changes in the operating system behavior at runtime. A key lesson learned is that there is a plethora of practical issues that must be resolved when doing this research. First, the noise signatures for the two machines we looked at are very different from each other. Second, the implementation used for the dynamic scheduler may not be most efficient, and requires more rigorous acceptance tests to check that it works. Third, weighted scheduling does make a difference as it can dynamically adjust weights when it is clear that some processors are slower than others, possibly because they are affected by high-frequency noise events. The knowledge of which processing elements are more pre-occupied with OS events can be used to reduce the chance that statically scheduled work being executing by some core will always finish later than that of the other cores. By finding the right point at which to adjust weights, we can provide more accurate and effective weighting scheme.

For future work, we plan to further experiment with measurement-based load balancing, possibly using Charm++. We hope that periodic load balancing will provide significant advantages for certain applications and can further reduce dequeue overheads. We also hope to make our auto-tuning more sophisticated and make choices such as the average tasklet granularity and tasklet granularity distributions automatically. By using more advanced heuristics for auto-tuning, we will be able to make our solution more portable for a range of existing and future architectures.

Acknowledgments

The authors would like to thank Maria Garzaran, Laxmikant Kale, and Franck Cappello for their input and

suggestions for this paper. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy (DOE) award DE-SC0004131. It was also supported in part by the U. S. Department of Energy grant DE-SC0001845 for HPC Colony II. Runs on Ranger were done under the TeraGrid [21] allocation grant ASC050040N supported by NSF. This research also used Jaguar which is a resource of the National Center for Computational Sciences at Oak Ridge National Laboratory, and is supported by the Office of Science of the U. S. Department of Energy under Contract DE-AC05-00OR22725. Accounts on Jaguar were made available via the Performance Evaluation and Analysis Consortium End Station, a DOE INCITE project. This document was released by Lawrence Livermore National Laboratory for an external audience as LLNL-CONF-492091.

References

[1] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[2] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 303–312.

[3] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[4] A. Blanchard, "Practical experiences with OS jitter," 2010. [Online]. Available: http://www.ibm.com/developerworks/wikis/display/LinuxP/OS+Jitter+Mitigation+Techniques

[5] S. Oral, F. Wang, G. Shipman, D. Dillow, R. Miller, D. Maxwell, J. Becklehimer, and J. Larkin, "Reducing application runtime variability on jaguar xt5," in *CUG '10: Proceedings of the 2010 Cray User Group Meeting*. Cray User Group, 2010.

[6] V. Kale and W. Gropp, "Load balancing for regular meshes on SMPs with MPI," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 229–238.

[7] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '96. New York, NY, USA: ACM, 1996, pp. 318–328.

[8] A. Bhatele, L. Wesolowski, E. Bohm, E. Solomonik, and L. V. Kale, "Understanding application performance via micro-benchmarks on three large supercomputers: Intrepid, Ranger and Jaguar," *International Journal of High Performance Computing Applications (IJHPCA)*, 2010, http://hpc.sagepub.com/cgi/content/abstract/1094342010370603v1.

[9] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 356–368.

[10] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003.

[11] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[12] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[13] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[14] S. Krishnan and L. V. Kale, "Automating Runtime Optimizations for Load Balancing in Irregular Problems," in *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, San Jose, California, August 1996.

[15] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[16] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *23rd ACM International Conference on Supercomputing*, 2009.

[17] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –12.

[18] A. Morari, R. Gioiosa, R. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *IEEE Parallel and Distributed Processing Symposium, 2011*, 2011.

[19] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, pp. 3–16, March 2008.

[20] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.

[21] C. Catlett *et al.*, "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," in *HPC and Grids in Action*, L. Grandinetti, Ed., vol. 16. Amsterdam: IOS Press, 2007, pp. 225–249.