Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality

Vivek Kale*, Simplice Donfack†, Laura Grigori†, William D. Gropp*
*
University of Illinois at Urbana-Champaign, Urbana, IL 61801
† INRIA-Saclay, Paris, France

***Abstract*— Performance irregularities on massively parallel processors lead to load imbalances and a significant loss of performance. Multi-core nodes suggest a promising way to redistribute work within a node, thus mitigating performance irregularities. However, there exists a non-trivial cost to redistributing work, and associated data, across cores. We investigate how work can be equitably distributed across cores without significantly disturbing data locality, and without incurring significant scheduling overhead. Towards this end, we design a series of scheduling strategies and tuning mechanisms; our foundational technique is an intelligent blending of static and dynamic scheduling. We also implement a basic runtime system and library to minimize programmer effort in applying these strategies. Our techniques provide 28.16% performance gains over static scheduling and 17.13% gains over guided scheduling for a widely used regular mesh benchmark, and 44.45% gains over static scheduling and 13.06% gains over guided scheduling for an n-body simulation, both on 1024 nodes.**

I. Maintaining Load Balance and Locality

Load imbalances come from the application (typically coarse-grained, persistent) or the architecture (typically fine-grained, transient). Figure 1 shows the timeline of a single execution of an already highly optimized dense matrix factorization code, which we refer to as CALU, on a multi-core node. In the timeline, load imbalances across cores occur due to the white areas of the timeline, which indicate idle times on a core. Work could be re-distributed across cores during execution time to reduce these idle times. The purpose of Figure 1 is to show that load balancing is an important and non-trivial problem for HPC applications, not just due to increasing complexity of applications, but also due to increasing complexity of architectures; load balancing is important even for computations that have conventionally been considered to not need load balancing, with this dense Communication-avoiding LU factorization being one example.

A basic load balancing technique, e.g., dynamic scheduling [1], can incur a cost of data movement to re-distribute work across cores, and this cost can significantly degrade performance of an application [2]. We also note that the significant costs of cache misses only increase for a multi-core node with a larger number of cores [3]). To minimize this cost, locality-aware scheduling [1], [2], [4] can be used. Figure 2a shows the application execution timeline for CALU when locality-aware scheduling (called "scheduling with locality tags") is used, and Figure 2b shows the application execution timeline when CALU uses a different type of locality-aware scheduling (called "scheduling with locality tags and stealing"). The basic locality-aware scheduling strategy and the optimized locality-aware scheduling strategy is explained in [2], and its application to the CALU code is discussed in [4]. The green rectangular block is the execution timeline of CALU. The cache misses during execution of CALU code (with the locality-aware scheduling technique applied) are in the red line graph below this execution timeline. In this case, there are several cache misses, indicating loss of locality and causing large performance degradations due to the significant costs of cache misses. The purpose of these graphs is to explain the problem that the strategies developed for this particular example may not be useful in another application or architecture, where the balance needed between locality and load balance may be different. With this, to balance the tradeoff between load balance and locality, our approach is to use an intelligent blend of static and dynamic scheduling, with the proportion of static and dynamic scheduling used being carefully tuned.

The two diagrams in the bottom of this section, which each contain three rectangular solids, show how our strategies are applied to an MPI+OpenMP 3D stencil code. The diagram on the left shows a single timestep of a 3D stencil code written in MPI+OpenMP, where work within each MPI process is statically scheduled across cores. The diagram on the right shows the application of our lightweight scheduling technique to this 3D stencil code, where the basic static/dynamic scheduling strategy is used. The strategy is defined to the left of these two diagrams. We note that the static fraction is the *scheduling parameter* of mixed static/dynamic scheduling. We call this strategy mixed static/dynamic scheduling, or Lightweight Scheduling.

II. Implementation and Scheduler Optimization

Optimizations over the mixed static/dynamic scheduling strategy can help to further balance the tradeoff between load balance and locality. Examples, as shown in the bottom-left quadrant, are constrained staggered static/dynamic scheduling and mixed static/dynamic scheduling with variable-sized tasks. Each strategy allows for new opportunities to help find the best balance between load balance and locality. Additionally, we must define the methodology for tuning the balance between load balance and locality. We can do this through experimental tuning, where we run the scheduler with different scheduler parameter values, and use the best-performing parameter values during application execution. We can also adjust the scheduler parameters at runtime, based on information from previous application timesteps. Finally, we can use model-guided optimization, developed through performance modeling

and theoretical analysis, during application execution to prune the search space for efficiently finding the best-performing scheduler parameters.

An illustration of the slack-conscious scheduling strategy, along with its corresponding formula used in its mechanism of model-guided optimization to tune the scheduler parameters, is shown in the right column. The formula shown is used to tune the static fraction per MPI process, based on MPI slack. We note that we assume one MPI process is assigned to a node of a cluster, as is done in [2], [3]. The slack is shown in orange in the figure, and is denoted by S in the formula shown. Figure 5 shows the code for a conjugate gradient (CG) computation transformed to use our technique. The static fraction used for the threaded computation region shown, enclosed by "FORALL_BEGIN(...)" and "FORALL_END(...)", is calculated through the use of the function call `predict_static_fraction()`.

III. Within-node Results and Analysis

The purpose of Figure 3 is to show the competitive performance of the mixed static/dynamic scheduled version of CALU for widely used implementations of LU factorization such as MKL and PLASMA. Our mixed static/dynamic approach provides 30.34% gains over the PLASMA library and 34.46% gains over the MKL library, and is further discussed in [3].

The timeline shown in Figure 4a shows the CALU computation as it is applied to static/dynamic scheduling. This timeline should be compared to the timeline in Figure 1. This shows how mixed static/dynamic scheduling improves both load imbalance. The histograms in Figure 4b show the distribution of execution times for 5000 independent executions of the CALU code on a single node of an Intel Westmere cluster. The distribution of execution times show impact to scalability, as discussed in [3]; having both absolute performance and performance variations are necessary for obtaining performance gains at large-scale. The performance of the executions in fully static scheduling is multi-modal, showing the impact of different load imbalances. The load imbalances come from both the application and the architecture. The performance variations are small for dynamic scheduling, but because of the overheads that dynamic scheduling incurs, performance degradation increases. Both the performance variations and absolute performance are least when mixed static/dynamic scheduling is used.

Table 1 and Table 2 show the performance improvement of our scheduling technique over OpenMP static scheduling on the NAS benchmarks for an Intel Westmere 16-core node and IBM BG/Q 16-core node, respectively. As can be seen in Table 1, the gains for the CG benchmark are large on the Intel Westmere machine due to the scheduler's handling of application load imbalance of NAS CG along with load imbalance due to performance irregularities sourcing from OS noise. As seen in Table 2, while the BG/Q machine has low noise [3], the scheduler still acheives significant gains for CG due to its ability to handle the application load imbalance of CG.

IV. Across-node Results and Scalability

Figure 6a shows the performance for different strategies as applied to the MPI+OpenMP regular mesh code SNAP [5]. Figure 6b shows performance for different scheduling strategies applied to an MPI+OpenMP n-body particle galaxy simulation code from the Rebound application [6], as we increase the number of nodes used in a cluster of SMPs. The speedups over static scheduling are shown. We focus on the results for figure 6b. The OpenMP *dynamic* scheduling strategy does only slightly better at small node counts, but helps at larger node counts where load imbalance across cores is more. The OpenMP *guided* scheduling strategy [7] does better than the *dynamic* scheduling strategy. The *uSched* strategy, which is the basic mixed static/dynamic scheduling strategy illustrated in the end of the top-right quadrant of the poster, greatly reduces the scheduling cost. This strategy helps both at small and large node counts. We note that *uSched* performs better than *guided* scheduling, due to *guided* scheduling not being able to maintain locality across timesteps, and due to its significant dequeue overheads for the large tasks. Additional results are in [3]. The *uSched* strategy provides 25.52% performance gains over static scheduling at large node counts. The *callsite* strategy is the optimization to the *uSched* strategy that takes into account MPI slack [8] to reduce scheduling overheads, and is illustrated in the bottom-left quadrant (diagram of slack-conscious scheduling strategy along with the theoretical analysis). This strategy provides 28.58% performance gain. The *vSched* scheduling strategy, which is an optimization to the basic *uSched* strategy that attempts to improve spatial and temporal locality, and that is explained in [9], provides higher gains throughout. When we combine the *vSched* and *callsite* scheduling strategy optimizations, i.e., *full* scheduling, we get an overall 44.45% performance gain.

References

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[2] 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.

[3] S. Donfack, L. Grigori, W. D. Gropp, and V. Kale, "Hybrid static/dynamic scheduling for already optimized dense matrix factorization," ser. IPDPS 2012, 2012.

[4] S. Donfack, V. Kale, L. Grigori, and W. D. Gropp, "Improving data locality for communication-avoiding dense matrix factorizations," ser. INRIA-Illinois Jointlab workshop, 2010.

[5] (2014, Jun.) Snap proxy application. [Online]. Available: http://www.lanl.gov/projects/feynman-center/technologies/software/snap-sn.php

[6] H. Rein and S.-F. Liu, "REBOUND: An open-source multi-purpose N-body code for collisional dynamics," *A&A*, vol. 537, p. A128, 2012. [Online]. Available: http://dx.doi.org/10.1051/0004-6361/201118085

[7] 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.

[8] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: making dvs practical for complex hpc applications," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 460–469.

[9] V. Kale, A. P. Randles, V. Kale, and W. Gropp, "Locality-optimized scheduling for improved load balancing on smps," *EuroMPI'14*, vol. 0, pp. 1063–1074, 2014.