Brief Announcement: Locality-aware Load Balancing for Speculatively-parallelized Irregular Applications

Youngjoon Jo and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
{yjo, milind}@purdue.edu

ABSTRACT

Load balancing is an important consideration when running data-parallel programs. While traditional techniques trade off the cost of load imbalance with the overhead of mitigating that imbalance, when speculatively parallelizing *amorphous data-parallel* applications, we must also consider the effects of load balancing decisions on locality and speculation accuracy. We present two *data centric* load balancing strategies which account for the intricacies of amorphous data-parallel execution. We implement these strategies as schedulers in the Galois system and demonstrate that they outperform traditional load balancing schedulers, as well as a data-centric, non-load-balancing scheduler.

Categories and Subject Descriptors: D.3.4 [Processors]: Run-time Environments

General Terms: Languages

Keywords: Speculative parallelization, Irregular programs, Data partitioning, Load balancing

1. INTRODUCTION

A common source of inefficiency in data-parallel programs (where elements from an iteration space are processed in parallel) is *load imbalance*, where different threads perform different amounts of work. Load imbalance can arise because one thread is assigned more work than another, or because the work assigned to one thread takes longer (due to, *e.g.*, locality effects). A number of load balancing schedulers have been proposed to minimize the effects of load imbalance, ranging from dynamic schedulers that allocate work on-demand [10] to work-stealing schedulers that allow threads without work to "steal" work from other threads [1, 2].

In recent work, we have identified a more general form of data-parallelism that arises in many irregular programs, called *amorphous data-parallelism* [5]. An amorphous data-parallel application is organized around a worklist of *active nodes*, which represent computation over a shared data structure. To process an active node, a portion of the data structure, called the node's *neighborhood* is read or written; this execution may generate new active nodes, which are added to the worklist. Parallelism arises by processing active nodes with disjoint neighborhoods in parallel. Active nodes with overlapping neighborhoods are said to *conflict*, and cannot be processed in parallel. In general, these applications must be parallelized using speculative parallelization [8].

Amorphous data-parallel algorithms differ from traditional data-parallel programs in notable ways. First, the iteration space may not be fixed: work may be added to the worklist during execution. Second, because the applications operate over irregular data structures, adjacent elements in the worklist may not exhibit locality. Finally, and most importantly, elements in the worklist may not be independent of each other; there may be cross-iteration dependences that must be respected.

Existing load balancing techniques, which have focused on traditional data-parallelism, do not perform well when faced with amorphous data-parallel programs. Load balancing schedulers suited for amorphous data parallel programs must address additional factors beyond run-time overhead and load balance: they must account for newly created work, how that work may be assigned to improve locality, and how load balancing decisions may impact misspeculation rates. Absent these considerations, a load balancing scheduler may deliver good load balance, but at the cost of overall performance.

In this brief announcement, we present two novel, *data centric* load balancing algorithms, *dynamic partition allocation* and *partition stealing*, as well as a data-centric version of the traditional work-stealing algorithm. These approaches account for the effects of speculatively parallelized, amorphous data-parallel programs. We implement our schedulers in the Galois system. Across three benchmarks from the Lonestar benchmark suite, we find that data-centric load balancing can improve performance by up to 66% over a non load-balancing scheduler, and by up to 25% over the best load balancing schedulers that do not account for locality effects.

2. BACKGROUND

2.1 The Galois system

The Galois system uses speculative techniques to parallelize amorphous data-parallel applications [8]. The system uses data partitioning to improve locality and reduce speculation overhead [7]. The data structure used in a program is partitioned (typically, into more partitions than threads). To minimize conflict detection overhead, two active nodes are considered to conflict if their neighborhoods lie in the same partition. It is hence critical to minimize the chance that two threads work on active nodes with neighborhoods in the same partition. To achieve this, partitions are mapped to threads and ac-

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.
ACM 978-1-4503-0079-7/10/06.

tive nodes from a partition are executed by the thread that owns the partition. This minimizes misspeculation, as only neighborhoods that cross partition boundaries can trigger conflicts. This *static, partition-based* scheduler is the default scheduler used by the optimized Galois system.

2.2 Load balancing techniques

A common load balancing strategy is *guided self scheduling* [10]. All the work is placed in a centralized worklist, and work is handed out to threads in chunks that decrease in size as computation progresses. While this approach improves load balance without incurring too much overhead from accessing the central worklist, it does not address locality. Furthermore, because the chunks of work are created in a partition-agnostic manner, it is quite likely that different threads will receive active nodes from the same partition, increasing misspeculation (as their neighborhoods will necessarily conflict).

An alternate approach to load balancing is *work-stealing* [2]. A work-stealing scheduler is organized around a set of *deques*, one per thread. Each thread takes work from the front of its deque, and places any newly generated work at the front, as in a stack. If a thread runs out of work, it steals work from the *back* of a randomly chosen deque. Thus, locality is enhanced during normal operation by processing newly generated work immediately, while steals preserve locality by removing work from the back of the victim's deque.

There are a few drawbacks to workstealing in the context of amorphous data-parallelism. First, if new work is rarely created, workstealing can lead to poor locality: the initial assignment of work may either be random, or use *binary splitting* [1], which places all the work in one worklist and distributes work through steals. In either case, there is no guarantee that locality will be attained. Second, steals potentially increase misspeculation: if a thief steals work from the same partition that another thread is working on, only one of the two threads will be able to make forward progress.

3. DATA-CENTRIC LOAD BALANCING

To address the problem of load imbalance in amorphous data-parallel programs without sacrificing locality or increasing misspeculation, we must take a *data-centric* approach to load balancing. We present three schedulers that leverage partitioning information when making load-balancing decisions: a variant

Partition-aware work-stealing: A simple strategy is to add partition-awareness to the work stealing scheduler. The scheduler adopts a data-centric model, using the same initial distribution of active nodes as the default Galois scheduler. Until steals start to occur, the application will enjoy the same locality as the static scheduler. Unfortunately, because steals do not consider partitioning information, they may still suffer from the misspeculation effects discussed in the previous section.

Dynamic partition allocation: In dynamic partition allocation, active nodes are associated with partitions as in the static scheduler. However, rather than assigning every partition to a thread prior to execution, the partitions are all placed in a centralized worklist. When parallel execution begins, and any time a thread runs out of work, it retrieves a new *partition* from the central worklist. As a result, the steady state of the scheduler is for a single partition to be assigned to each thread and the remainder of the partitions to be in the central worklist. Each thread treats the set of active nodes in a single partition as a stack, with newly generated work in that partition being processed immediately.

This scheduler attempts to gain the benefits of dynamic scheduling without incurring its costs. There are fewer accesses to the centralized worklist, as work is distributed at the partition granularity. Furthermore, work is handed out in locality-preserving chunks, reducing misspeculation.

Partition stealing: The final data-centric scheduler we investigate is a variant of work-stealing. The initial distribution of work to threads is partition-based, as in the static partitioned scheduler, and in partitioned work-stealing. However, rather than stealing individual pieces of work—or larger, but still essentially random chunks—when a thread runs out of work it attempts to steal a *partition* from another, randomly selected, thread. When a thread is operating in its normal, non-stealing mode, it behaves as in the static partitioned scheduler. This scheduler attempts to mitigate the misspeculation effects of workstealing by ensuring that each thread works in separate partitions.

4. EVALUATION

Methodology: We evaluated three benchmarks from the Lonestar suite of irregular programs [6]: Delaunay triangulation (DT), Delaunay mesh refinement (DMR) and preflow push (PFP). For each benchmark, we evaluated 6 different load balancing schedulers: (i) the default static, data-centric scheduler, used as the baseline for our comparisons; (ii) *self*, a scheduler using guided self-scheduling; (iii) *steal*, a workstealing scheduler using binary splitting; (iv) *steal**, a workstealing scheduler that assigns initial work using partition information; (v) *dynamic(P)*, a dynamic partition-allocation scheduler; and (vi) *steal(P)*, a partition-stealing scheduler.

Our schedulers were implemented in the Galois system, using partition-locking [7]. The system, schedulers and applications were written in Java 6 and executed on the Sun HotSpot VM version 1.6 with a 12GB heap. To account for the effects of JIT compilation, each configuration was run 10 times, and the average of the two median run-times was recorded. All of our experiments were conducted on a Sun Niagara 2 server, consisting of two 8-core chips in an SMP configuration.

Results: Figure 1 shows the performance of the six load-balancing schedulers, relative to the static scheduler. The non-data-centric schedulers are plotted with solid lines, while the data-centric schedulers are shown with dashed lines.

DT does not suffer from significant load imbalance, as new work is not generated; on 32 threads, the average time each thread is idle when using the static scheduler is only 13%. As such, the primary goal of a load balancing scheduler should be to preserve locality and performance. Unfortunately, *self* and *steal* perform poorly, because they do not consider partitioning information. However, introducing data-centrism improves performance: *steal(P)* performs similarly to the default static scheduler, and outperforms the best non-data-centric scheduler by up to 9%.

DMR has more load imbalance, as new work is generated; the mean idleness with the static scheduler is 21% on 32 threads. Again, we see that by ignoring locality effects, the non-data-centric schedulers perform poorly, while the data-centric schedulers perform well. The best data-centric scheduler, *dynamic(P)*

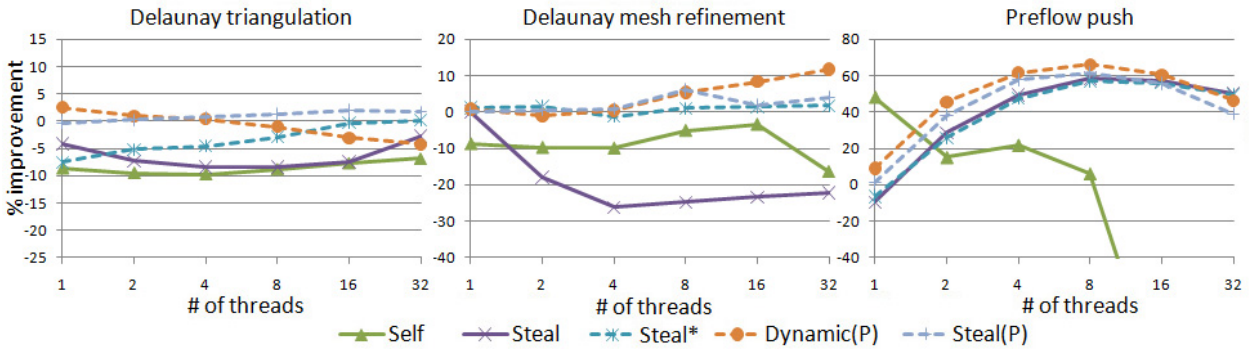

Figure 1: Performance of load balancing schedulers relative to Galois default.

outperforms the best non-data-centric scheduler, *self*, by up to 24%, and outperforms the static scheduler by up to 12%.

PFP has the worst load imbalance: the static scheduler has an average mean idleness of 58% on 32 threads, and 69% on 8 threads. Because most of the work in PFP is generated during execution, the initial distribution of work has little bearing on performance, and *steal* and *steal** exhibit similar behavior. The two other data-centric schedulers outperform the work-stealing schedulers up to 16 threads, at which point overheads and abort ratios cause their performance to become comparable. All the load balancing schedulers, save *self*, substantially outperform the static scheduler. *Dynamic(P)* outperforms the static scheduler by up to 66%, and outperforms the best non-data-centric scheduler, *steal*, by up to 25%.

5. RELATED WORK & CONCLUSIONS

Related work: Markatos and LeBlanc identified the tension between schedulers that attempt to improve locality and schedulers that attempt to improve load balance, and noted that most schedulers target load balance at the expense of locality [9]. They noted that in many cases, it is more important to concentrate on locality, even if that results in poorer load balance. Our work echoes their finding in the context of amorphous data-parallelism, where locality-aware schedulers outperform schedulers that do not pay heed to data-locality concerns.

Recent studies have investigated extending work-stealing to consider locality. Krishnamoorthy *et al.* investigated a locality-aware work-stealing scheduler for sparse-matrix computations [4]. However, these applications do not use speculation or generate new work, unlike our target algorithms. Guo *et al.* studied a work-stealing scheduler for X10 [3], which preserves locality by limiting a thread's steal targets to other threads working on local data.

Conclusions We introduced three *data-centric* schedulers: a partition-aware variant of work stealing, a data-centric dynamic scheduler and a data-centric work stealing scheduler. Across three benchmarks, we find that our load balancing schedulers outperform the default, locality-ware scheduler by up to 2% on Delaunay triangulation, 12% on Delaunay mesh refinement and 66% on preflow push. Furthermore, on each benchmark, our data-centric schedulers outperformed the best non-data-centric schedulers by up to 9% on Delaunay triangulation, 24% on Delaunay mesh refinement and 25% on preflow push. Clearly, making load balancing decisions with an eye toward data-centrism is critical to achieving high performance on amorphous data-parallel programs.

References

[1] *Intel Threading Building Blocks Reference Manual, Rev 1.9*. 2008.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[3] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 341–342, 2010.

[4] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, and P Sadayappan. An approach to locality-conscious load balancing and transparent memory hierarchy management with a global-address-space parallel programming model. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[5] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2009. ACM.

[6] Milind Kulkarni, Martin Burtscher, Keshav Pingali, and Calin Cascaval. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, April 2009.

[7] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 233–243, New York, NY, USA, 2008. ACM.

[8] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[9] E. P. Markatos and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. Technical report, Rochester, NY, USA, 1991.

[10] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), 1987.