A Case for a Situationally Adaptive Many-core Execution Model for Cognitive Computing Workloads

Masab Ahmad*, **Chris J. Michael**†, **Omer Khan***
*University of Connecticut, Storrs, CT
†Naval Research Laboratory, Stennis Space Center, MS

***Abstract*—Cognitive computing has emerged as a challenge application domain that requires sensor-to-decision algorithms. However, situational dynamic changes in such algorithms give rise to efficiency challenges in computational settings. These algorithmic variations stem from *input dependence*, such as the context of data or the sparsity of the graph being processed. Consequently, concurrency control becomes challenging since the complex data-dependent behavior in these workloads exhibits a range of plausible parallel implementations. Moreover, today's computational hardware increasingly exposes concurrency controls to the software, and the right choice for the execution model varies with the algorithm and its input characteristics. In this paper we address the question of how to efficiently harness concurrency controls in the cognitive computing context. We present a situationally adaptive scheduler (`SAS`) that manages input dependence to yield a near-optimal parallelization strategy at the software layer, and the right concurrency control at the architecture layer. We evaluate `SAS` on a large-scale simulated multicore and show that situational scheduling helps achieve better scalability over naive settings.**

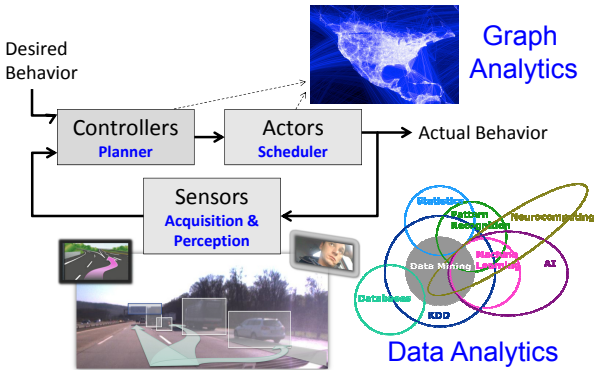

Fig. 1. A simplistic view of the cognitive computing "loop". The complexity of the loop is exacerbated by the spatial and stochastic variables as well as temporal integration of inputs.

I. Introduction

The society in general is experiencing new trends towards cost efficiency and reliance on machines for everyday tasks. Our daily lives are increasingly becoming dependent on computational platforms with advanced cognitive skills. Such cognitive platforms require augmented *sensor-to-decision* processing [6], whereby they continuously sense their environment and compute their decisions using a computational substrate that may reside onboard or offline (e.g., a datacenter). We consider a simplistic view of the cognitive computing "inner loop" that must sense, control and act on sensed input. Figure 1 shows the application domains for the respective tasks i.e., sensor data processing and perception, task coordination and scheduling, and path planning. The algorithms/heuristics that represent these applications fall in the category of data and graph analytic problems. They operate on unstructured data and exhibit complex dependence patterns that are known only during program execution [8]. Therefore, advancements in data and graph analytics give rise to stringent concurrency requirements.

On the computational front, the industry has aggressively adopted multicore technology that itself exposes concurrency controls to the software. Even though increases in processor efficiency and memory density help scalability, computational resources in current systems still tend to be severely over- or under-utilized, leading to non-optimal efficiency. Dynamic input changes [22] [12] lead to performance consequences as well that create issues in scalability and throughput. We can take an example of an autonomous vehicle (AV) tasked with finding a single source shortest path (SSSP) in a dynamic environment. Sensor data is input to weather models, which produce graphs that are mapped onto an architecture to be processed to make decisions for shortest path calculations. Situational changes in weather, location, and other events cause changes in input graph characteristics. This leads to challenges in architectural requirements to maintain efficiency and real-time guarantees for the AV, primarily because different algorithms provide different scalability for different input types. This results in the so called "Ninja Performance Gap" [25], which is the humongous performance gap between programs naively mapped onto architectures and programs optimized for efficiency.

So why do these "Ninja Performance Gaps" occur? The primary reason is that situational changes in inputs lead to challenges from software through architecture. As depicted earlier, data and graph analytic algorithms are inherently irregular and exhibit complex data-dependence [8]. Inputs come in various types with changes occurring primarily due to situational dynamic alterations, and different algorithmic parallelizations scale depending on these inputs provided to the program. These situational changes in software also propagate to the underlying computational hardware. As processors get more heterogeneous and parallel (e.g., Intel Xeon Phi, Graphic Processing Units (GPUs), and the brain-inspired IBM TrueNorth chip), variations caused by situational settings expose more performance gaps within underlying architectures.

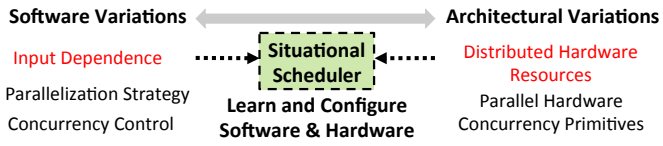

Fig. 2. Situational Choices and Situational Scheduling.

Such variations may also lead to choices that perform best when executed sequentially, hence creating a range of single- through multi-threaded mappings. This leads to challenges in architectural mapping, such as providing optimal thread scalability and resource utilization. In these situational contexts, end-users and programmers do not fully consider these implications of their programs and end up implementing sub-optimal concurrency strategies at both software parallelization and hardware level concurrency controls.

Finding an optimal combination in such situational settings involves correlating large sub-spaces of software and architectural configurations from which one combination must be chosen. A situational computing model is thus needed that understands correlations, harnesses all available choices, and optimally configures sensor-to-decision processes of the cognitive "inner loops" efficiently onto software and hardware.

We formulate the proposed situational scheduler (SAS) as an artificial intelligence (AI) problem, where a learner trains based on some offline data/information, then solves a choice search space, as shown in Fig. 2. One positive aspect of using an AI learner is that it can improve over time and deliver results with high accuracy. However, the learner needs to articulate all situational choices, which is a hard problem due to the large space of combinations. The problem becomes even harder as more choices are added to the computational paradigm, which increases software complexity due to choices between various available libraries and for various machine types. We demonstrate a prototype implementation of SAS for graph analytic algorithms executing on a large scale simulated multicore. We analyze situational paradigms, such as input dependence and concurrency control, and show efficient parallel execution and scalability.

II. Related Work

Prior works have made many attempts to solve this adaptive problem generically for various architectures and algorithms. Autotuning works such as OpenTuner [4] and PetaBricks [3] statically search spaces of program optimizations to find a software combination that performs best on an underlying architecture. Such works do not assume any prior knowledge of which combinations are preferable (which makes our work orthogonal to autotuners), and thus are constrained by their learning time (several tens of hours) due to program search spaces (with complex combinations of up to 10^{3657} combinations for workloads such as Poisson) [10]. Synthetic inputs can be used as plausible training data to reduce complexity in such cases [16] while maintaining ample evaluation accuracy [17]. Other works, such as automatic parallelization compilers [24] [7] statically assemble programs for concurrency, while other frameworks, such as HeartBeat [13] and SEEC [14], optimize for performance dynamically. However they do not take into account situationally dynamic input changes together with choosing which algorithm or parallelization might work best on an algorithm or architecture. Either way, a holistic approach to cater for cognitive computing paradigm still remains an unsolved problem.

III. SAS Prototype

This section motivates the need for a situational scheduler in cognitive settings. We show how input dependence causes variations in concurrency requirements and how these requirements affect performance. Then the proposed scheduler (SAS) framework is formulated. Finally, the evaluation shows how the scheduler performs in a simulated multicore setup.

A. Situational Choices

One of the most ubiquitous graph problem falls in the domain of finding a single source shortest path (SSSP). Cognitive platforms, such as UAVs and self-driving cars use SSSP as a core algorithm for path planning. We therefore take SSSP as a challenge problem to explain what choices are available that need to be exploited to ensure performance optimality. The SSSP choices are executed on a simulated 256-core multicore using the Graphite simulator [11]. We use the Graphite simulator because many-core chips with hundreds of cores do not exist yet. Each core is modeled as an in-order pipeline with $32KB$ private L1 instruction and cache caches, and a $256KB$ shared L2 cache. The 256-core processor also models eight memory controllers to access the off-chip memory. All input graphs for SSSP problem have an adjacency list representation.

Algorithm 1 A Generic Graph Algorithm Skeleton

```
1: for (Each vertex u) do                    ▷ Outer Vertex Loop
2:     for (Each Edge of u) do               ▷ Inner Edge Loop
3:         Do Parallel Work (Locks may be required)
4:     Do more Parallel Work (Sync threads if required)
```

1) Input Dependence: Input graphs come in many different types, sizes, and compositions, depending on situational settings. All graph algorithms have *outer loop* that traverses the graph's vertices, and *inner loop* that traverses the neighboring edges of a given vertex, as shown in the generic graph skeleton in Algorithm 1 [15] [8]. Input dependence influences graph sparsity and density, which determines how many inner loop iterations are required per outer loop. Graphs with less inner loop traversals have more sparsity, and vice versa. Due to continuously evolving inputs (cf. Figure 2), the situationally changing graphs lead to choices in parallelization strategies that exploit either edge-level parallelism in the inner loop, or vertex level parallelism in the outer loop. This algorithmic template is generic to almost all graph algorithms, and thus these two parallelization strategies with respect to the graph's sparsity apply to a larger domain of graph analytics [25].

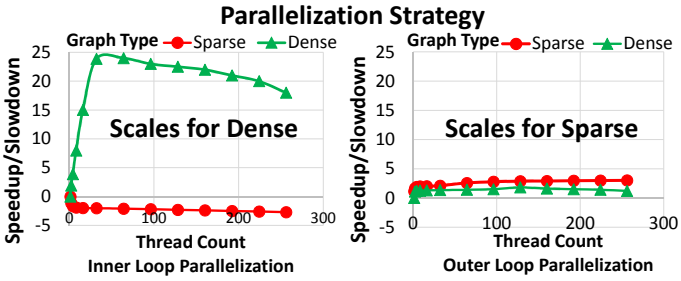

Fig. 3. Parallelization strategies for SSSP. Dense graph has 16K vertices, 8K edges per vertex and Sparse graph has 16K vertices, 16 edges per vertex

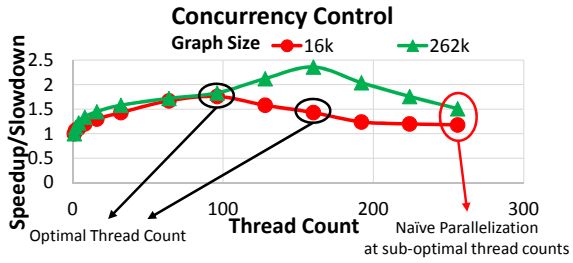

Fig. 4. Concurrency Control Issues for SSSP in Multicores. Sparse graph used.

Sparse graphs perform best with outer loop parallelizations, while dense graphs work well with inner loop parallelization strategies.

2) Parallelization Strategy at the Software Layer: In case of the SSSP algorithm, Figure 3 shows results for different parallelization strategies, where speedup or slowdown is computed by comparing completion time at various concurrency setups over the sequential SSSP. We observe that the edge-level (inner loop) parallelization performs best for a dense graph that has higher connectivity between neighboring vertices, while the vertex-level (outer loop) parallelization strategy works best for a sparse graph that has less connectivity. It is noteworthy that selection of inner loop parallelization strategy for sparse graph results in a slowdown when compared to the sequential execution. This makes it even more important to choose the right parallelization strategy situationally. In general, input dependence and parallelization strategies are correlated in a sense that parallelization strategies are a software consequence of input dependence. To fully harness efficiency in such situational settings, these choices must be availed properly.

3) Concurrency Control at the Architecture Layer: Figure 4 shows how two different sparse graph sizes observe different scalability patterns, while already exploiting the efficient vertex-level parallelism. This results as an architectural consequence due to input dependence, where choices are now exposed within the underlying hardware as optimal thread count. A large graph scales to higher thread count, while a smaller graph scales less. This happens because a larger graph encompasses more work that can be distributed amongst threads, while small graphs encompass less work, and hence less parallelism is available. Shortcomings, such as inter-algorithmic dependencies and synchronization bottlenecks also reduce scalability at high thread count. These issues allow some algorithms to scale only to intermediate thread counts, while some scale to larger thread counts, depending on the available parallelism. Due to lack of knowledge of scalability patterns, a naive parallelization would try to use the maximum number of threads available in the multicore processor, and hence not perform optimally. A scheduler is thus required that predicts scalability and then controls concurrency in such situational settings.

B. SAS: Situationally Adaptive Scheduler

Using the choices described earlier, we formulate the proposed situational scheduler that caters for software and hardware using input situations. Various inputs and architectural combinations combine into an extremely large number of possible configurations. Given such a high complexity problem, it is imperative to reduce the overall software and hardware search space. Heuristics that are primarily associated with machine learning are useful in such a setting.

Machine learning algorithms, such as neural networks or support vector machines, can deliver optimal search results with low complexity. At a higher abstraction, supervised machine learning is widely acknowledged as a scalable and efficient solution to problems with large sub-spaces [18]. The downside of using such algorithms is that they require some offline training data to be tuned and optimized to find optimal results. However in the case of our scheduler, offline data can be readily and easily available. Parameters such as available core counts on the target machine, parallelization libraries, and some information on scalability can further help supplement the learning process.

1) SAS Framework Flow: Initially, training data that is generated offline is input to the machine learning scheduler, so it can learn about the available algorithms and machine parameters. This training data contains performance numbers such as speedups obtained, and/or other required parameters obtained from a slew of synthetic inputs. Some a priori information on which configurations perform better on specific machine setups is also assumed to be within this training phase. Such supervision reduces learning complexity, and is expected to normalize configuration to perform optimally.

Once the scheduler is fully tuned, the user feeds the SAS scheduler with the desired algorithm and its input characteristics. Such inputs can be real inputs such as road networks, or changing weather graphs which are obtained over time. The scheduler then analyzes the given input and run it through its machine learning algorithm. Once the user defines inputs, the learner then starts the evaluation phase, where output configurations are generated, and output accuracy is determined. Using machine learning, SAS picks optimal parallel algorithm and the concurrency setup, which are then deployed on the target machine. For now we are targeting only performance related parameters using SAS for multicore setups, but additional parameters such as energy, accuracy, resilience, security, and additional machine types,

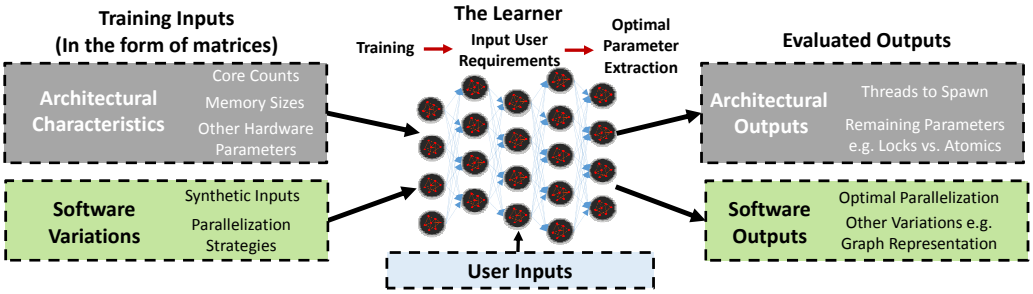

Fig. 5. High Level Learner Architecture.

will be added in the future.

2) The Learner: In this paper, SAS is implemented using the Multi-layer Perceptrons (MLP) based neural network [9], as shown in Figure 5. MLP is well suited since it effectively captures non-linear input dependencies and scalability patterns in situational choices. Larger AI systems are also starting to use MLP blocks as their central computational paradigms, primarily due to its ability to be molded with respect to the application, as well as scalability. Such machine learning programs are also easily parallelizable, which is expected to be helpful for problems involving larger complexity. Hardware support for MLP-like systems is also showing promise, with IBM's Truenorth chip being an example. This is just one prototype implementation for SAS, we expect to evaluate other learning and even control-theoretic frameworks as future work.

MLP consists of multiple layers of neurons, each layer with a finite number of neurons and associated sigmoid functions to capture non-linear characteristics of the training inputs. Input layer neurons represent varying inputs for a given algorithm, such as core counts, available parallelization strategies, and different input types. Output layer neurons represent parameters, such as optimal parallelization strategy and thread count for a given input configuration. There are also intermediate layers, which perform the actual learning. The more layers and neurons a network has, the better choice classification accuracy is achieved, however with growing complexity.

The proposed MLP system uses four layers of neurons, with each layer having 16 neurons, along with a final reduction step that outputs the required parameters. Various MLP configurations with different neurons per layer are tested, the results of which are available in the evaluation section. It is found that learning accuracy and classification performance saturates at around 96% for the given neural network parameter setting.

IV. Methodology

Parallel graph analytic benchmarks are taken from the CRONO suite [1], namely **SSSP**, **PageRank**, **BFS**, **DFS**, and **Community**. CRONO allows interfacing these algorithms with synthetic and real world graphs. Even though other graph and data analytic workloads can be evaluated easily, we leave them for future work due to space constraints. All benchmarks are executed on a simulated 256-core multicore, as described in Section III. Performance is measured as speedup or slowdown by comparing the completion time at various concurrency setups over the sequential version.

Random synthetic graphs are generated using a modified version of the GTgraph generator [5]. All graphs use adjacency list representations. SAS is trained using synthetic input graphs, and then evaluated using the sparse California Road Network from the SNAP directory [19], and a Connectomics graph [21] that is a dense map of neural connections within a human brain. It is assumed that the programmer has little know how about the underlying software and hardware architecture, and makes several sub-optimal naive choices. We evaluate two naive settings as baselines.

Naive Parallelization is an easy to program parallelization strategy, similar to the one used in Parallel MiBench [15]. The programmer chooses to exploit edge level parallelism, which is easier to program as compared to vertex level parallelism that requires dynamic scheduling of vertices for efficiency. To keep this setting restricted to parallelization strategy, the programmer uses thread count that gives best performance, similar to SAS.

Naive Threads chooses an optimal parallelization strategy but a sub-optimal thread count, as depicted in Fig 4. We assume the programmer naively executes the program on the maximum number of threads available on the target machine.

Next, we evaluate and compare SAS with the naive settings.

V. Evaluation

Figure 6 shows the relative speedups for CRONO benchmarks using the California road network [19], which is a sparse input graph. Both naive parallelization and thread choice strategies show limited scalability. However, SAS observes a speedup of 3.5× on average over naive parallelization setting. In road network graph each vertex is connected to 1.5 vertices on average. Therefore, the edge level parallelization strategy has limited work to be distributed per thread. Since threads in graph benchmarks are synchronized often using fine-grain communication, this leads to limited scalability due to lack of exploitable parallelism. Vertex level parallelization on the other hand opens up pareto fronts of multiple vertices per iteration. This constitutes more parallel work that SAS is able to exploit efficiently by picking the right parallelization strategy. We also observe that even naive parallelization executing at the optimal thread count cannot match the scalability offered by the right parallelization strategy. However, SAS is

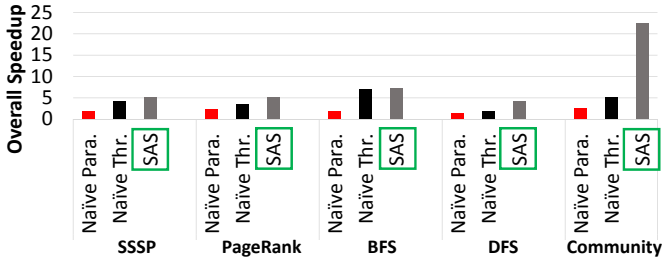

Fig. 6. Speedups observed using SAS for the California Road Network.

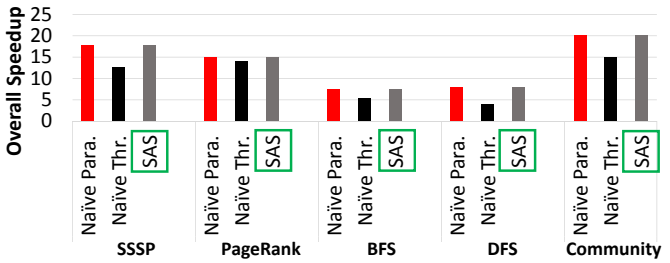

Fig. 7. Speedups observed using SAS for a Dense Graph.

able to learn and adapt this input dependence and achieve superior scalability and performance. For the naive thread setting comparison, we observe an average performance improvement of 2× for SAS. This advantage is solely achieved due to the optimal thread count choice. The main reason is that several of the graph benchmarks (e.g., **DFS** and **Community**) scale up to an intermediate thread count, and after that the communication costs overwhelm the parallel computations. As a result the communication and synchronization costs result in slowdowns relative to the optimal thread count. Since SAS learns the near-optimal thread count based on the input dependence, it picks the right thread count and delivers better performance compared to the native threads setting.

Figure 7 shows the relative speedups using Connectomics brain graph, which is a dense input graph. The edge level parallelization strategy works best for dense graphs since they exhibit enough exploitable parallel work that can be distributed among the threads. Therefore, we observe that naive parallelization setting yields similar results as SAS. In this graph input type, SAS improves performance by choosing the optimal thread count. Due to disparities between optimal thread count and naive threading, similar to Fig 3, performance improvements for SAS are on average 30%. Moreover, dense input graphs have more compute component than communication, so the overall results are less effective as the work done per thread is already reasonable for all thread counts. **SSSP** and **Community** show larger improvements than other benchmarks because their communication costs augment at a higher rate at high thread counts, which SAS is able to avoid by scheduling them at lower than maximum threads.

Overall, SAS is able to adapt to input sensitivity by adjusting the parallelization strategy and concurrency control, with minimal complexity. This learning framework reduces programmer level dependence by automating choices a programmer would have otherwise deployed manually, for efficiency.

Learner accuracy is important since it defines the speedup acquired over naive settings. The number of neurons per layer are therefore varied, and speedups in contrast to an ideal baseline are computed. Figure 8 shows the evaluation accuracy of the learner, with respect to changes in the number of neurons, where the accuracy saturates for larger number of neurons. However, large space requirement increases memory, computations, and overall complexity. It is therefore imperative to manage trade-offs between classification accuracy and acquired performance. In our implementation, we get ∼96% accuracy at 16 neurons, which provides plausible performance for most applications.

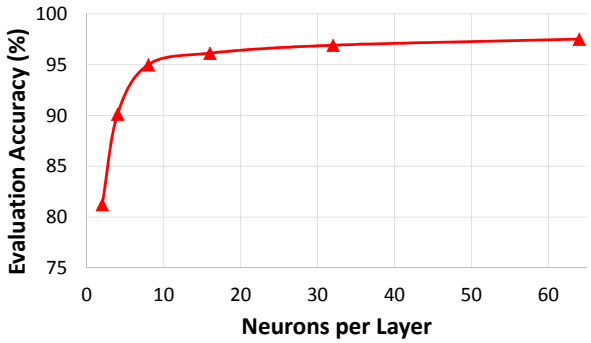

Fig. 8. SAS accuracy for various neural network configurations. 4 layers of neurons used.

VI. Discussion

A. Improvements to the Scheduler Framework

Sensor-to-decision problems can be ported to various machine types, such as GPUs and IBM TrueNorth chips. The proposed SAS prototype generates optimal parameters for a large-scale simulated multicore setup. Under machine variations, our a priori assumption regarding machine information and scalability is expected to help with optimal choice configurations. Training data for such setups can be generated and applied to SAS to schedule applications.

B. Emerging Concurrency Control Methods

In the context of concurrency control, emerging communication acceleration primitives within architectures can be readily applied to SAS. Prior works have shown that programming models, such as pushing store data model [23], processing in memory model [2], and conventional shared memory atomics based model, all work well. However, the right choice of a communication primitive depends on the algorithm, its input type, as well as the machine's concurrency architecture. SAS can learn these choices and use a target programming model that exploits communication scalability and efficiency.

C. Additional Computational Metrics

A possible future direction is to consider the implications of security for cognitive computing. With continuously evolving cyber-security threats it is almost impossible to protect everything in a computational system that requires performance

constraints. For example, cognitive applications leaking information can be made oblivious in software such that they mitigate leakage of private data [20]. However, these oblivious algorithms incur large performance overheads. SAS can utilize oblivious libraries only when vulnerabilities increase under certain situations. This will improve performance and help cognitive systems such as UAVs to meet real-time guarantees.

Resilience is also an important metric for cognitive computing. Soft errors can lead to disastrous consequences in various cognitive applications, such as in a UAV flying at high altitudes. However, just like security, soft errors do affect all aspects of the application code equally, and thus cognitive applications can be selectively protected [26]. Resilient libraries, alongside conventional algorithm libraries can be integrated into SAS, which can select optimal algorithms based on the overall system parameters, such as the location and altitude of a UAV.

VII. Conclusion

Sensor-to-decision applications are becoming more and more ubiquitous in our daily lives. However, such workloads suffer from sub-optimal implementations, primarily due to dynamic input changes. To circumvent this problem, we present SAS, a situational scheduler that learns algorithm implementation and architecture parameters using offline training input data and schedules optimal thread count and/or parallelization strategy for a many-core system. We motivate SAS using the single source shortest path (SSSP) problem, which not only serves as a longstanding cognitive challenge problem for graph analytics but is also ubiquitously used in real world applications. We show that our situational scheduler learns optimal choices and improves performance over naive parallelization strategy and concurrency control. For sparse input graphs, we observe 2–3.5× improvement. However, for dense graph the naive strategies perform reasonably well. Even in this case SAS improves performance by an average of 30% over naive settings. These improvements show that SAS is a plausible framework solution for cognitive applications.

References

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono : A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proc. of IEEE Int. Symposium on Workload Characterization*, ser. IISWC, 2015.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 105–117.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, no. 6, pp. 38–49, Jun. 2009.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316.

[5] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," 2006.

[6] A. Bulu and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[7] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "Helix: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 84–93.

[8] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IEEE Int. Symposium on Workload Characterization (IISWC)*, Sept 2013, pp. 185–195.

[9] R. Collobert and S. Bengio, "Links Between Perceptrons, MLPs and SVMs," in *Proceedings of the 21st Int'l Conference on Machine Learning*, ser. ICML, 2004.

[10] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proc. of the 36th ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, ser. PLDI 2015. NY, USA: ACM, 2015.

[11] J. E. M. et. al., "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan 2010, pp. 1–12.

[12] J. Gao, C. Zhou, J. Zhou, and J. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *Data Engineering (ICDE), 2014 IEEE 30th Int. Conf. on*, March 2014, pp. 556–567.

[13] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88.

[14] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. E. Miller, S. M. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. P. Chandrakasan, and S. Devadas, "Self-aware computing in the angstrom processor," in *49th Annual Design Automation Conference*, ser. DAC '12, 2012.

[15] S. Iqbal, Y. Liang, and H. Grahn, "Parmibench - an open-source benchmark for embedded multiprocessor systems," *Computer Architecture Letters*, vol. 9, no. 2, pp. 45–48, Feb 2010.

[16] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman, "Synthetic data and artificial neural networks for natural scene text recognition," *CoRR*, vol. abs/1406.2227, 2014.

[17] G. Karsai, A. Ledeczi, J. Sztipanovits, G. Peceli, G. Simon, and T. Kovacshazy, *Self-Adaptive Software: Applications: Second International Workshop, IWSAS 2001, Hungary, May 2001*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ch. An Approach to Self-adaptive Software Based on Supervisory Control, pp. 24–38.

[18] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24.

[19] J. Leskovec and et al, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[20] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *Security and Privacy (SP), 2015 IEEE Symposium on*, May 2015, pp. 359–376.

[21] D. Mhembere and et. al., "Computing scalable multivariate glocal invariants of large (brain-) graphs," in *Global Conf. on Signal and Information Proc. (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 297–300.

[22] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 145–156.

[23] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim, "Location-aware cache management for many-core processors with deep cache hierarchy," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, 2013, pp. 20:1–20:12.

[24] D. Prountzos, R. Manevich, and K. Pingali, "Synthesizing parallel graph programs via automated planning," in *Proc. of Int'l Conf. on Programming Language Design and Implementation*, ser. PLDI 2015, 2015.

[25] N. Satish and et. al., "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. of the 2014 ACM SIG. Int. Conf. on Management of Data (SIGMOD)*. NY, USA: ACM, 2014.

[26] Q. Shi, H. Hoffmann, and O. Khan, "A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 85–89, July 2015.