Title
Exploration and Analysis of Memory-aware Semi-Static Scheduling Algorithms

URL
https://github.com/yingyee0111/memory-aware-scheduling

Summary
Our project aims to explore the scheduling of parallel tasks. More specifically, we aim to study parallel scheduling algorithms that take in account both task execution time and memory accesses. Our project will have two main objectives. First, we will construct a framework/process for testing the effectiveness of various scheduling algorithms through actual program simulation as well as multi-core cache memory simulations. Secondly, we will use this framework to analyze a few scheduling algorithms/metrics in order to characterize their performance. Ideally, the system we would propose would first perform an initial task scheduling based on static parameters for any arbitrary parallel program. Memory activity (reads/writes) and overall execution time would be tracked for each task while they are being executed. Then, at certain synchronization points in the parallel program, the system would perform synchronization, and reschedule the tasks based on the profiled memory accesses and execution time. This would hopefully allow us to reduce the overhead of memory accesses and increase workload balancing, while minimizing the performance overhead. An actual implementation of such a system in real time for a real program would be a stretch goal of the project.

Background
Modern CPUs make use of caches in order to hide the latency of accesses to main memory, by storing relevant or previously accessed data in the cache which can be more quickly accessed by a CPU core than main memory. However, for multicore processors with shared memory, caching behavior can be more complicated. Memory may be loaded into the cache for one core, but not the other. In addition, changes to data from one core may need to be propagated to other cores in order to maintain a coherent memory system. This is done through some of the cache coherence techniques we learnt in class. Other techniques such as hardware prefetching as well as virtual memory further complicate our conception of what goes on in a cache.

However, when we look at the performance of parallel code, it is important to take into account memory accesses since memory operations and communication are often the bottleneck of parallel code. Therefore, a system that is able to take these operations into account when scheduling parallel tasks has the potential to greatly increase the performance of parallel programs.

The cache hierarchy can vary greatly from processor to processor. For example, many processors we see in consumer laptops and desktops have 3 levels of cache (L1/L2 for each cores and L3 shared between cores). However, certain multi-core low power/embedded processors only have 2 levels of cache (L1 for each core and L2 shared) while other CPUs even

have up to L4 cache. In addition, the L1 cache is sometimes split into instruction/data caches to store instruction memory and data memory separately. Note that for this project, we will not be considering the instruction cache. Therefore, any memory-aware scheduling algorithm would have to be conscious of the cache configuration of the system it is running on.

Our proposed system will make use of semi-static scheduling in order to reduce the overhead of the scheduling algorithm, since it is likely to be expensive to compute. In addition, less frequent rescheduling attempts allows us more time to profile the individual parallel tasks while they run, giving us a better understanding of their memory accesses and execution time. Understanding the computation time allows us to balance the workload better when the workload of tasks is uneven. Understanding memory accesses allow us to schedule tasks that share memory onto the same core. Since tasks on a single core share caches, memory fetching overhead can be saved if previous tasks have already loaded the data required for subsequent tasks.

The Challenge

The scheduling problem is challenging as it is in itself an NP-hard problem, our scheduler can only use heuristics to give an approximately good allocation. Adding the memory access factor also makes the problem more difficult as we are optimizing with 2 different metrics. Our allocation has to achieve an even distribution across cores based on the computation time per task, and also optimize for tasks with memory sharing to be placed on the same core. We have to choose which metric to prioritize and adjust the weight given to each metric. For instance, we can prioritize balancing computation time by adopting a greedy scheduling algorithm on tasks sorted by computation time, but pick the one with the most memory sharing amongst the next 5 candidates.

We also have to consider the tradeoff of rescheduling as it will definitely increase the runtime. Adjustments would include changing the frequency of rescheduling and choosing cheaper scheduling algorithms that may give worse scheduling.

Another problem we have is that we will never have perfect information. Profiling the runtime and memory accesses before the rescheduling point can only give us a rough estimate of the actual values. Hence, the effectiveness of rescheduling also depends on how well the profiling reflects the actual behavior of the future.

Resources

First, we will select certain benchmark parallel programs in order to use to conduct our analysis. These programs might involve code previously written in this class (such as in Assignment 3), as well as open-source parallel benchmarking code available online.

Our front end of the code base will be started from scratch where we will decompose programs into parallel task segments. At each rescheduling point, we will profile the computation time and memory access thus far. The computation time will be estimated based on real-world execution time of individual parallel tasks as well as the number of instructions executed. The memory access estimations would be done with the aid of the cache simulator. This cache simulator

would be adapted from the starter code from 15-346. https://github.com/bprail/cadss_public. The total runtime of the parallel program will be evaluated with the help of the computation time and memory access times estimated. This data will be used to analyze the performance of various scheduling algorithms.

We will be referencing the following papers as well as others for memory aware scheduling algorithms.

M. Gonthier, L. Marchal and S. Thibault, "Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems," 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 694-704, doi: 10.1109/IPDPS53621.2022.00073.

Maglalang, Jordyn; Krishnamoorthy, Sriram; and Agrawal, Kunal, "Locality-Aware Dynamic Task Graph Scheduling" Report Number: (2016). All Computer Science and Engineering Research. https://openscholarship.wustl.edu/cse_research/1167

Chunlin Li, Jianhang Tang, Hengliang Tang, Youlong Luo,
Collaborative cache allocation and task scheduling for data-intensive applications in edge computing environment, Future Generation Computer Systems, Volume 95, 2019, Pages 249-264, ISSN 0167-739X, https://doi.org/10.1016/j.future.2019.01.007.

If time permits, we will insert code simulating the scheduling system into our benchmark parallel programs. This scheduling code would attempt to tie threads to specific cores (likely using the sched_setaffinity instruction) according to the scheduling algorithm, in order to evaluate the performance of our scheduling system in the real world. We hope to run this code first on smaller multicore machines, before moving on to the PSC machines.

Goals and Deliverables

Plan to achieve

- Decompose benchmark programs into parallelizable tasks. Identify a fair suite of programs and annotate the task units and synchronization points.
- Profile computation time and memory access at rescheduling.
    - To profile computation time, we will be generating assembly instructions, estimating the clock cycles needed, as well as actually running the individual parallel tasks
    - To profile memory accesses, we will extract the addresses used in stores and loads.
- Explore various memory aware scheduling algorithms and assess their effectiveness.
- Build a cache simulator to estimate the memory access time.
- Evaluate the total run time (computation time + memory access time) of our program with and without dynamic scheduling.

Hope to achieve

- Run the actual program on local multicore and PSC machines to evaluate how the scheduling algorithm performs outside of our simulated environment.

Demo plan

- It would be difficult to demo our scheduling algorithm, since the results are based on objective performance metrics and analysis, and may be hard to visualize. If our performance gain is very impressive (such as double the speed), we might be able to demonstrate the performance gain for the program in real time. However, if the performance gain is less impressive (which is almost certain), we will likely just show graphs and perhaps trace animations for our scheduling algorithm.

Platform choice

We will use C++ on the front end and C on the backend (cache simulator). Programs will be run on PSC machines if we have sufficient bandwidth. We will specify which tasks to run on which core according to the scheduling algorithm.

Schedule

Week	Planned Work
Apr 1 - Apr 7	Understanding the cache simulator, begin work on extending cache simulator as needed for multiple cores of a specific cache hierarchy Survey the different scheduling algorithms from research papers and select a few promising ones.
Apr 8 - Apr 14	Finish up implementation of cache simulator, ready for testing. Begin work on front end of the program evaluation framework Work on computing the cache affinity of tasks from the memory access collected.
Apr 15 - Apr 21	Finish up the front end of the evaluator. Stamp out bugs in the evaluation platform found in testing. Run selected algorithms to schedule the tasks based on the profiling.
Apr 22 - Apr 28	Begin work on implementation of the actual scheduling algorithm in parallel code for real world testing. Evaluate selected algorithms based on performance we obtain from testing. Refine testing procedure and algorithms where necessary. Prepare deliverables: poster, demo and report
Apr 29 - May 4	Actually run the scheduling algorithm on parallel code on local multi-core machines, as well as the PSC machines if time permits