Schedule Data, Not Code

Micah J Best
University of British Columbia
mjbest@cs.ubc.ca

Shane Mottishaw, Craig Mustard, Mark Roth, Parsiad Azimzadeh, Alexandra Fedorova
Simon Fraser University
{smottish,cam14,mroth, paa4,fedorova}@cs.sfu.ca

Andrew Brownsword
Electronic Arts, Inc.
Vancouver, Canada
brownsword@ea.com

Abstract

In this paper we argue that the scheduler, as the intermediary between hardware and software, needs to be fully data-aware. The old paradigm of envisioning tasks as amorphous blobs of 'work' to be assigned to processors is incomplete and needs be expanded. Some techniques and projects have emerged that implicitly use this idea, but either focus on a small aspect of data or are targeted to optimizing specific problems. We argue for more general solutions.

1. Introduction

Anyone who has written a parallel program only to find that increasing the number of threads makes the program run *slower* understands the problem: there is more to parallel programming than assigning 'work' to 'processors'. Shared resources, bus traffic, cache effects and many other factors contribute to the program's runtime. In addition, many opportunities are often missed. SIMD units are left under-utilized by fragile auto-vectorizers, and cache-coherency protocols can cause an unavoidable eviction of soon to be needed data. There are now many great tools and techniques for decomposing code, protecting critical sections and other traditional concerns. However, the memory hierarchy of the modern computer has grown – not only in depth, but in width. Consider the increase in NUMA[1] architectures. The movement of data becomes a dance around various levels of cache and processing units, and the aggregate costs can overwhelm the savings of parallelization. Manually orchestrating this dance is difficult, time consuming, generally requires an expert and is rarely portable. Accounting for all the aspects of complex software running on a variety of complex systems is quickly becoming untenable. Conversely, simply accounting for some subset of software-system pairs yields a solution that can only prove fruitful in extremely presumptive scenarios. What can be done to aid programers, both expert and novice, to achieve faster parallel programs? Our position is that the scheduler should not only map code, but also data, to the machine's physical resources, thus reducing the costs of data movement. We call this methodology *Data-Informed Scheduling* (*DIS*).

Consider a program that is written to execute as a series of tasks. As with many programs, certain tasks can be interleaved whereas others may exhibit pairwise temporal dependencies, requiring that one task completes before another is invoked. A traditional scheduler will just assign tasks to processors, honouring said dependencies and perhaps attempting some load-balancing. Suppose we have a task that traverses a linked list. A *DIS* scheduler is provided with the knowledge that it is operating on a linked list and can thus begin prefetching the nodes of the list as soon as the task is launched. Furthermore, a *DIS* scheduler has the ability to place this task on processors that have the nodes that it operates on available in cache.

We will describe *DIS* and give evidence to back up our argument (Section 2). Many researchers have encountered the problems of data costs and have derived techniques with that in mind (Section 3). We will discuss (Section 4) how our suite, Cascade, goes beyond those projects and how *DIS* is informing its continued development. To add weight to our discussion we will give a video game based example of how *DIS* yields a

[1] Non-Uniform Memory Access. NUMA processors have local memories and the time access a particular memory item varies greatly depending on its location relative to the requesting processor.

performance improvement (Section 5). Finally, we conclude with a discussion of our future work (Section 6).

2. Overview of *Data-Informed Scheduling*

We begin this section with some concrete motivation for needing a *Data-Informed Scheduler* and proceed to give a high-level description of the *DIS* method. *DIS* is intended to be a set of principles to influence those working on parallel systems. We present these principles to influence design decisions and levels of abstraction in their work.

We focus heavily on application level scheduling, as this is our domain. However, this argument applies to all levels of scheduling.

2.1 Importance of Scheduling Data

So far, programmers have been fortunate that relatively flat memory hierarchies, prefetching, and cache-coherence protocols have largely obscured the cost and complexity of data migration. However this will not be the case in the future. Future multi-core hardware will have deep memory hierarchies, with high costs for inefficiently accessing memory. Hardware prefetchers are limited in their ability to make smart decisions, and some manufacturers are experimenting with removing cache-coherency protocols.

The cost of sharing data on modern systems is increasing. These costs include not only synchronization, but the movement of data across processors. The authors of Corey [4] report that on a 16-core AMD Barcelona system with four NUMA domains, the cost of accessing the L1 cache is only 3 cycles, while the cost of accessing a remote L1 cache goes up to 121 cycles, and finally the cost of accessing a remote memory node can reach as many as 327 cycles! Clearly we can save a lot of cycles if the data needed by the computation is placed physically close to the processor where the computation will run.

Trends in hardware suggest that the large discrepancies between the costs of local and remote memory accesses will continue to be present in future hardware. New processors increasingly use the NUMA architecture. Some specialized hardware, such as the Cell processor used in the Playstation 3, requires the programmer to explicitly place program data into the memory buffer of the SPE where the data will be processed [8]. Looking in the mobile space, we also see multicore designs and memory hierarchies with non-uniform access latencies and lack of coherence, requiring the programmer to explicitly place data into processor memories, so as to optimize access efficiency [1]

The onus to navigate this complexity should be on software. Specifically: languages, compilers, and runtime schedulers must be able to provide programmers with a productive toolkit to create programs that can execute on a wide variety of multicore hardware.

2.2 *Data-Informed Scheduling* Methodology

Contemporary application-level schedulers are limited in their capacity to understand data/code relationships. These schedulers are often FIFO task-based systems that are limited to assigning tasks to threads. The data use of each task is never available to the scheduler.

A scheduler that has such a limited representation of the program it executes can only assign blocks of machine code instructions to processors. The movement of data is implicit in these instructions and many opportunities for optimization are lost. Processors cannot consecutively schedule tasks that use the same data, copy a list into an array for SIMD processing just before it is needed or change a method of iteration to suit the current number of available processors. Even worse, the scheduler is unable to prevent potential pitfalls such as the storm of bus traffic that can be created by scheduling certain data-parallel tasks on too many processors. A conventional scheduler has no facility to make this kind of choice.

We have identified a number of program aspects that the compiler, language, and runtime stack should communicate to the scheduler in order for the scheduler to have a robust representation of the program. These aspects fall into four categories:

***What* data is used?** *Does the task modify global values? Does it affect a linked structure? Does it need only to read from a counter?* Without this information the scheduler cannot perform such optimizations as maximizing cache usage, and the scheduler must either trust that code makes safe concurrent access to data or must make conservative scheduling decisions.

***Where* does the data comes from?** *Is the data in main memory? Does it need to be retrieved from disk or other storage? Is this data the product of another task stored in a communication buffer?* Without this information the scheduler can schedule tasks with high latency costs in front of those with small latency costs.

***When* is the data used?** *What dependencies need to satisfied for this task to run? What state does the system need to be in before execution?* Without this information the scheduler cannot reflect the ordering of task. Understanding the prerequisites for tasks allows the scheduler to support such optimizations as critical path execution.

***How* is the data processed?** *Does the task perform a tree walk? Does it employ a formal parallel pattern [11]? Are its operations reorderable?* Many

common parallel patterns have multiple implementations that each work best in specific circumstances. A scheduler must be free to choose between these implementations.

These aspects (which we refer to later as *what*, *where*, *when* and *how*) are not directly implementable, but are more requirements that any parallel programming stack must provide in order to effectively schedule data. In order to practically leverage *DIS* principles, languages and compilers must be designed to provide and preserve the above semantic information and deliver it to the scheduler.

Finally, to make decisions based on these code/data relationships, a scheduler also needs a representation of the hardware on which its running. Scheduling two tasks that share the same data on two cores which share an L1 cache is quite different than scheduling these tasks on different chips. In order to make these choices, a scheduler need to be able to calculate these costs.

Determining the exact requirements that this hardware representation must provide is a challenge. The right balance must be struck between abstraction and usability. A scheduler that interpreted the original source code, maintained an exact map of the state of each cache and contained code to execute numerous parallel patterns and algorithms would be orders of magnitude too cumbersome to be effective. However, simpler representations might similarly be unusable. Adapting this representation to changing hardware will also be a concern.

In order to strike an effective balance between accuracy and overhead the scheduler will necessarily need to rely heavily on abstractions of both program and hardware. As we have shown in our previous work on Synchronization via Scheduling[3] and will show further in Section 5 even a very simple coarse representation can make a considerable difference.

3. Steps Towards *Data-Informed Scheduling*

Many projects and techniques have implicitly incorporated elements of *DIS* with positive results. Their success adds weight to our arguments and motivates further exploration and generalization of these techniques. A discussion of the entire body of work is unfeasible due to space constraints, but we will feature a small number of specific examples.

3.1 Map/Reduce

Map/reduce has great deal of potential for *DIS*. The specification of a map/reduce job carries with it crucial information about data access. For map tasks the data comes from a file and for reduce tasks it comes from the output of the map tasks (*where*). Processing is done by linear iteration with certain variations (*how*). First the data is mapped then its transformed version is reduced (*when*). Furthermore, map/reduce implicitly notifies the system how the data will be partitioned among processing nodes or cores and, consequently, their memories. The semantics of map/reduce imply that the data is partitioned linearly for map tasks and the size of the partition is determined by the number of map tasks.

Computation is expressed using a programming pattern with well-understood semantics. A map/reduce system is structured such that all data handling among units of computation is performed by the framework and not by the programmer. It is relatively straightforward to apply extremely powerful locality optimizations. Chen, Chen and Zhang observed that they could increase CPU cache efficiency if the output of the map task were immediately processed by the corresponding reduce task [6]. To implement this idea, called tiled map/reduce, they chopped the data into blocks that fit in the processor cache. Map/reduce was performed on these blocks in stages and merged at the end. As a result, they reduced memory requirements by as much as 85% and improved performance by up to a factor of 3 times. These optimizations would not have been possible if semantics of data access were buried inside conventional imperative code.

3.2 Chapel

Although map/reduce is a great example of how the aspects of *DIS* can benefit performance, it applies only to very specific types of computation. The designers of Chapel took this further and implemented a programming language with first-class support for data-centric constructs [9] (*what*, *how*). Chapel targets arrays and similar linear memory structures, such as hash tables. In Chapel, arrays are specified as domains and each domain has a corresponding map that specifies how the array is implemented. For instance, domain maps specify how domain indices and array elements are mapped to memory locales (e.g., nodes or cores), how they are stored in memory, and how operations, such as accesses and iterations, are implemented. Furthermore, Chapel provides native support for data partitioning. Providing a high-level specification of how the data is laid out and accessed enables the runtime system to place the data right where it is needed and right when it is needed, which can dramatically reduce memory latency.

3.3 OS Examples

To minimize the cost of moving data across chips (*where*), Tam et al. proposed *thread clustering*, where the OS clusters threads that actively communicate on the same chip by monitoring cross-chip communication [12]. Like in many OS examples, the optimization is performed without assuming any knowledge about the application. However, the authors also showed that

a minimal knowledge about application data access patterns (*how*) can enable a much simpler implementation which bypasses the monitoring phase. This added simplicity is a further reason why *Data-Informed Scheduler* should be explored.

The Tornado operating system approached the vision of *DIS* to a much closer extent [7]. In Tornado, key kernel data structures are implemented as clustered objects. Each instance of an object is associated with a hardware domain (i.e., a processor and a corresponding memory), and whenever an object is accessed, the request is directed to the right hardware domain in an efficient manner (*where*). Tornado demonstrated how a technique akin to the proposed *Data-Informed Scheduling* with the use of knowledge about data accesses can increase efficiency of memory accesses.

More recently, Corey [4] and Barrelfish [2] reaffirmed the need to address data scheduling on modern multicore NUMA systems, proposing solutions similar in spirit to Tornado. In a yet another example, Boyd-Wickizer et al. proposed a memory-aware design of a file system cache [5]. In the proposed system, cached file data is partitioned across chips, and a thread accessing a file object is migrated to the chip "owning" the object (*where*). The authors discovered that thread migration was justified only in cases where the file object was sufficiently large, so the saved cost of data movement outweighed the cost of thread migration.

While all these examples demonstrate concepts in the spirit of *DIS*, they each address a specific problem. Our goal is to develop a general solution that can be used to solve a large number of problems encountered in real applications.

3.4 Synchronization via Scheduling

While the previous examples of work related to DIS concentrate primarily on the movement and placement of data in the memory hierarchy, DIS is not limited to reducing data costs. One example of this is a technique called Synchronization via Scheduling [3]. SvS utilizes static and dynamic analysis to determine *what* data is accessed by a task, before it is executed. SvS uses this information not to reason about data movement, but rather to determine which tasks can be safely executed in parallel: if it is determined that task A and B both modify the same data, then it simply schedules A and B such that A completes before B is run. Compare this to a scheduler without SvS. In this case, the programmer would be forced to manually serialize tasks that access the same data, or use potentially expensive synchronization primitives. In contrast, SvS is able to *automatically* extract parallelism in the case of serialized tasks, while avoiding the costs of synchronization primitives (e.g. aborts in transactional memory).

4. Cascade

Cascade is our parallelization suite for video games that is being designed to embody the principles of DIS.

Cascade represents programs using a task-graph based model. In a task-graph model, nodes represent code that has been divided into discrete units called tasks and edges represent an ordering between two tasks. If there is an edge (A, B) then task A must complete before B can be executed. Therefore a task-graph provides information about *when* data will be accessed by a task. Cascade further augments this by providing explicit data-flow between tasks in that one task sends data to another task. Since data is sent by placing it in a communication queue, the scheduler knows *where* this data resides.

In order to facilitate writing programs based on the above program model, Cascade provides a new programming language, the Cascade Data Management Language (CDML). CDML and the static analysis it enables, is also required to implement SvS, described in section 3.4, which Cascade uses to determine *what* data tasks access. Beyond this, CDML provides expressions for parallel patterns, providing a rich set of information as to *how* data is accessed.

Finally, Cascade incorporates a simple representation of hardware where a map describes which processors share a common cache. If processors share a cache, we say they are neighbours. During work stealing, a thread executing tasks on processor **A** will attempt to steal tasks from a thread executing on **A**'s neighbour, thus leveraging cache locality.

5. Experiments

To illustrate the *Data-Informed Scheduling* methodology we will describe how we added a locality optimization to Cascade, and show how it benefits a realistic application. We modified the CDML compiler to automatically add metadata, an integer called an *affinity id*, to each object used in a data-parallel operation. Each affinity id corresponds to a core. When the operation is executed, the id of an object is exposed to the scheduler which uses it to determine which core to assign that object to. A task is then executed on the assigned core with the object as its input. During execution of that task, any other object read or written will be assigned the input object's affinity id. Affinity ids are changed at most once per frame.

Our test application, QuakeSquad is designed to reflect the critical aspects of a game while being simple enough to modify easily and show experimental results clearly, akin to SynQuake [10]. In QuakeSquad a number of AI agents, citizen and techs, navigate around a 2D world. Two more entities, bombs and loot, influence

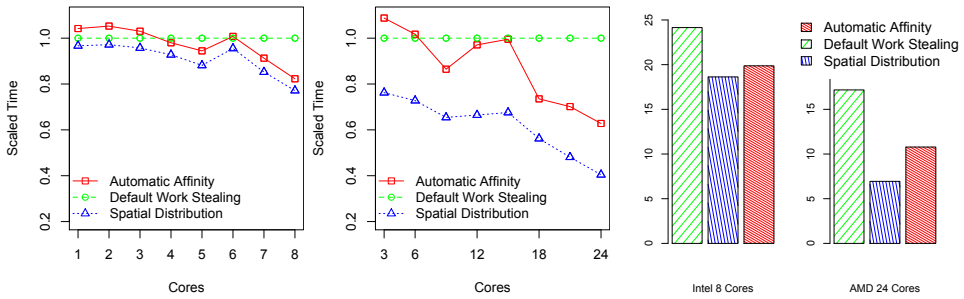

Figure 1. QuakeSquad Entity update performance: Relative (Intel *left*, AMD *center*) and Absolute (*right*)

the behaviour of the agents. Bombs cause citizens to flee and attract techs who will attempt to disarm them.

For each frame rendered, an update function for each entity performs multiple line of sight tests to each other entity nearby. These nearby entities are read, and the current entity is updated. This behaviour causes data of the entities that surround a particular entity to be brought into the same cache. This presents an opportunity to make locality optimizations to increase performance. In our experiments, we refer to this non-optimized behaviour as *Default*.

A popular optimization to 2D worlds is to spatially partition the world into cells where each cell is represented by a list of the entities within that cell. We implemented this partitioning, called *Spatial Distribution*, where we dispatched entire lists instead of single entities. This optimization achieves efficient locality optimizations and work-distribution, but required a great deal of hand-coding.

By enabling the affinity id tracking for the processing of entities we effectively produced an effect similar to the cell partitioning without modifying the program code. We call this technique *Automatic Affinity Grouping*. Determining appropriate times to automatically enable this optimization is part of ongoing research.

Our experiments were performed on an dual socket 8 core Intel Xeon E5405, and a 4 socket, 24 core, Opteron 8435. The results are shown in Figure 1. The left and centre graphs show the average frame time normalized to the unoptimized default. The right graph shows the average absolute performance in *ms* using the maximum cores for each machine.

At a small number of cores the overhead of Affinity Grouping outweighs its benefit. As the core count increases, performance of Affinity Grouping becomes faster than the Default by approximately 18% and 38% on the Intel and AMD respectively. While affinity grouping is slower than the hand coded spatial distribution, both similarly increase in their effectiveness as cores are added, and refactoring the code for spatial grouping was a major effort while affinity grouping simply involved enabling a feature in the scheduler.

6. Conclusion and Next Steps

We have described a methodology for building schedulers that can address the reality of the increasingly high cost of accessing data. This methodology included a description of several aspects that we believe are critical for the scheduler to understand about the program that it is running. The next step is to refine this methodology. Primarily we intend that the principles listed should be formalized into an actual model for *DIS* that is both actionable and can be reasoned about.

In future work, we intend to expand Cascade to further embody the principles of *DIS*. The affinity id technique is one of many that we are currently considering.

We hope to refine the work done by ourselves and others into a full, concrete specification for how parallel programs should interact with the underlying scheduler in order to realize practical benefits on tomorrow's increasingly complex hardware.

References

[1] Platform 2012: A Many-core Programmable Accelerator fro Ultra-Efficient Embedded Computing in Nanometer Technology. In *Whitepaper: STMicroelectronics and CEA* `http://www.cmc.ca/NewsAndEvents/Events/~/media/English/Files/Events/20101105_Whitepaper_Final.pdf`, November, 2010.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[3] Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: managing shared state in video games. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, pages 43–57, 2008.

[5] Silas Boyd-Wickizer, Robert Morris, Nickolai Zeldovich, and M. Frans Kaashoek. Managing Multicore Caches. In *Workshop on High Performance Transaction Systems*.

[6] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.

[7] Ben Gamsa, Orran Krieger, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *OSDI '99*, pages 87–100, 1999.

[8] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26:10–24, March 2006.

[9] Cray Inc. Chapel language specification 0.796 `http://chapel.cray.com/spec/spec-0.796.pdf/`.

[10] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM.

[11] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[12] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, 2007.