Insulating the Scientific Programmer from Perilous Parallel Architecture

Sean Chester
Computer Science
University of Victoria
Box 1700 STN CSC
Victoria, BC, Canada
schester@uvic.ca

Celina Gibbs
Computer Science
University of Victoria
Box 1700 STN CSC
Victoria, BC, Canada
celinag@cs.uvic.ca

Fil Rossi
Electrical and Computer Engineering
University of Victoria
Box 1700 STN CSC
Victoria, BC, Canada
filrossi@gmail.com

Andrew Brownsword
Electronic Arts Blackbox
4330 Sanderson Way
Burnaby, BC, Canada
brownsword@ea.com

Poman So
Electrical and Computer Engineering
University of Victoria
Box 1700 STN CSC
Victoria, BC, Canada
poman.so@ece.uvic.ca

Aaron Gulliver
Electrical and Computer Engineering
University of Victoria
Box 1700 STN CSC
Victoria, BC, Canada
agullive@ece.uvic.ca

ABSTRACT

The resource requirements of modern-day scientific applications is making hardware acceleration a requisite design consideration. But typically these applications are created and implemented by scientists trained in other fields, not the intricacies of parallel processing. Consequently, there is a definite need for abstractions.

In this paper, we discuss why the typically object-oriented abstractions that are commonly made available in such settings introduce accidental sequentiality and render the hardware acceleration ineffective. We hence propose a few research directions that we intend to pursue in order to generate new abstractions that better insulate the programmer from implementation details that do not relate to application logic.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction*

General Terms

Design, Performance

Keywords

abstraction, parallelization, scientific computing

1. HIGH PERFORMANCE SCIENTIFIC COMPUTING

Scientific computing is uniquely positioned where the demands for high performance converge with an especial need for layers of abstraction. It is not atypical that a scientific application requires weeks or months to run, even after rigorous optimisation, simply as a result of its computationally intense nature. It is quite an imposition for a scientist developing such an application to additionally develop the substantial system-level expertise required to really thoroughly optimise for a specific hardware architecture. At any rate, these optimisations might not transfer to the evolution of parallel architectures in coming years that could include hundreds of cores.

This process is not terribly unlike spending the time to research and open an investment portfolio, only to abandon it and start over when a brighter opportunity inevitably arises. At some point, the returns on the efforts are diminishing in contrast to simply allowing the first investment to flourish. And, anyway, one could have had a broker overseeing it and saved the time for research nearer to his passion and expertise. It is anticipated that most programming of future parallel systems will be done at the abstraction level of domain specific languages [1], such as Matlab. Even for mainstream programmers developing customized scientific applications, this indicates that parallelism will be supported by lower-level mechanisms, the details of which are insulated from the programmer. To circumvent the potentially laborious rigor of low-level optimisation and the difficulties of retaining these optimisations as code bases are ported across emerging architectures, support mechanisms should be in place to insulate the programmer from the particularly salient architectural considerations such as memory layout. It is crucial, too, that these considerations are not lost by these mechanisms.

Historically, programming languages that allow for high-performance array manipulation, such as Fortran and C are the languages of choice for many of the scientific community's numerical applications. This is understandable because of the conceptual similarity between sequential memory and arrays.

But, scientific computing is no longer limited to just applying the numerical algorithms implemented in popular numerical libraries such as IMSL,[1] LAPACK,[2] and GNU Scientific Library.[3] In fact, these libraries are just collections of building blocks for creating complex scientific programs, not unlike the insulation offered by objects. For instance, a typical scientific application would include a GUI and a computationally intensive solver that uses numerical libraries. The complexity inherent in such an application necessitates the abstractions provided by libraries and objects.

2. ITERATING ALL OVER THE HEAP

The conflict, however, comes from the disconnect between the abstractions in which programmers are used to thinking and the architectural needs of the hardware on which they are being applied. Favoured object-oriented patterns can really destroy the efficiency of the code, when the efficiency is the true purpose of using the hardware in the first place. A fantastic example is an iterator: it offers a conceptually clean way to walk through the elements of a collection, and is therefore a first choice for many OO programmers. But, it implies order within the collection that is not necessarily there, unintentionally sequentialising the programmer's code. If the code was meant to be run sequentially, anyway, this is fine; but, in a parallel setting, the task independence has been obscured from the compiler.

Even the simplest of data structures needs a review. Arrays are used in a variety of contexts. Contrast an array implementation of a set to storing a time series in an array. In the former case, there is complete independence between the components of the array, and in the latter, there is complete dependence. As a consequence, the compiler cannot predict whether the elements of the array can be treated as embarassingly parallel or not. Thus, although the data structure itself does not necessarily imply sequentiality, nor can one assume from its use that something as simple as a parallel-for pragma can be applied.

3. AN ARCHITECTURE-CONSCIOUS APPROACH TO ABSTRACTION

Rossi's experience [4] illustrates the cost of embracing traditional abstractions first, rather than considering the strengths of the architecture from the outset. In order to approach theoretical throughput on the NVIDIA GPGPUs, the data layout needs to be coalesced, or an order of magnitude penalty is incurred. (See Figure 1, taken from Rossi [4].) Abstractions that cannot coalesce the data, then, automatically pay this penalty, so it is important to consider the data layout from the stage of algorithm design. By not considering data layout from the perspective of the architecture initially, first the application takes an efficiency hit, and then, second, the programmer takes an efficiency hit when (s)he has to redesign the application. Consequently, one should adopt an approach of fitting problems to architecture, rather than trying to optimise algorithms that were written agnostic to their parallelisation needs.

[1]`http://www.vni.com/products/imsl/`
[2]`http://www.netlib.org/lapack/`
[3]`http://www.gnu.org/software/gsl/`

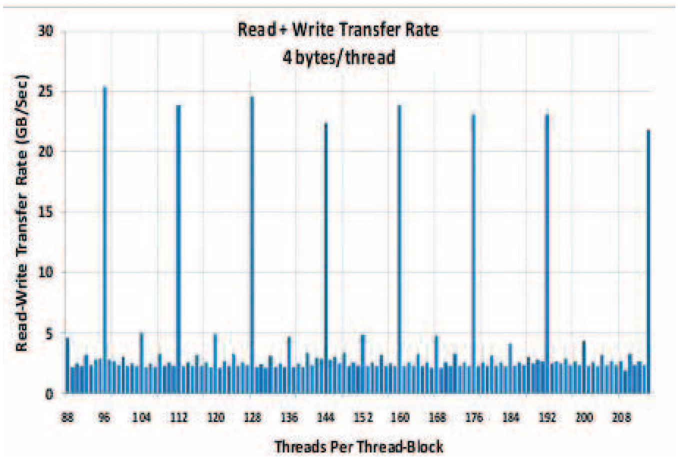

Figure 1: An illustration of the effect that coalesced/aligned data layout has on transfer rate on an NVIDIA GPU. The thread-block size is varied and throughput between the CPU and GPU memory stores is measured. Note the order of magnitude difference that arises simply out of data layout. (Taken from Figure 6-2 in Rossi [4].)

4. CONCLUSION AND RESEARCH DIRECTIONS

Scientific programmers need abstractions. But the abstractions typically used unintentionally sequentialise the execution of their code, squandering the potential that the architecture provides. Thus, new architecture-centric, rather than programmer- or object-centric, abstractions are necessary.

To this aim, our research directions are toward easing the incorporation of architectural insight into the design of data structures and algorithms—identifying the abstractions and design patterns that do not introduce such accidental sequentiality. In this manner, the scientific programmer can be truly insulating from the perils of programming parallel architecture.

At a higher level, this involves research into architecture-type-centric design patterns like SIMD, MIMD, &c., as in [2, 3]. Additionally, developing theoretic cost models to evaluate parallel algorithms based on the type of architecture on which they are designed to run, permits one to evaluate competing designs rigorously.

At a finer granularity and in the immediate future, we're starting with a couple case studies. The first is of sorting algorithms on the IBM CBE[4] and on the NVIDIA/CUDA GPGPUs[5] to identify which characteristics of the algorithms match the architectures' needs. The second is a similar approach with a couple of data mining algorithms on the Amazon EC2 cloud.[6]

[4]`http://www.ibm.com/developerworks/power/cell/index.html`
[5]`http://developer.nvidia.com/object/gpucomputing.html`
[6]`http://aws.amazon.com/ec2/`

5. ACKNOWLEDGMENTS

This research is supported by IBM and MITACS Canada.

6. ADDITIONAL AUTHORS

Additional author: Yvonne Coady (Computer Science, University of Victoria, email: `ycoady@cs.uvic.ca`)

7. REFERENCES

[1] Catanzaro, B., Fox, A., Keutzer, K., Patterson, D., Su, B.-Y., Snir, M., Olukotun, K., Hanrahan, P., and Chafi, H. Ubiquitous parallel computing from berkeley, illinois, and stanford. *IEEE Micro 30* (2010), 41–55.

[2] Gardner, H. Design patterns in scientific software. In *Computational Science and Its Applications (ICCSA)* (2004), Springer/Verlag, pp. 776–785.

[3] Mattson, T. G., Sanders, B. A., and Massingill, B. L. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[4] Rossi, F. Graphics hardware accelerated transmission line matrix procedures. Master's thesis, University of Victoria, 2010.