

Thalweg: A Framework For Programming 1,000 Machines With 1,000 Cores

Adam L. Beberg
Department of Computer Science
Stanford University
beberg@cs.stanford.edu

Vijay S. Pande
Department of Chemistry
Stanford University
pande@stanford.edu

Abstract

While modern large-scale computing tasks have grown to span many machines, each with many cores, traditional programming models have not kept up with these advancements, resulting in difficulty exploiting these computing resources with only modest programmer effort. Thalweg seeks to address this breakdown in several ways. It provides a model for designing algorithms that have the potential to scale to multiple cores and machines, with subsequent optimization by software engineers. Based on this concept, Thalweg presents an API for handling these algorithms, for transferring data to and from nodes and coprocessors, and for verifying the correct operation of the hardware. Finally, Thalweg presents a set of concepts and a laboratory framework for pedagogical use that will educate the next generation of software engineers to operate in a world in which multi-core and distributed computing are everywhere.

1. Introduction

Developing software that runs efficiently on systems that span thousands of cores is an extremely complex process, since one must deal with both the domain specific challenges of an application as well as the complexity of programming distributed and multi-core architectures. Thalweg is an extraction of the common elements of this problem from nearly two decades spent developing distributed and multi-core algorithms. Thalweg gives the domain expert and the software engineer each their clear role in the process of developing highly optimized software, and solves several other problems, such as making the resulting software more flexible for when new hardware becomes available.

Throughout this manuscript, is envisioned the example of a researcher or someone in industry that has access to a cluster or large multi-core device. This researcher is not an expert in the details of that device,

and would much rather spend their time doing their research, but wishes to use the device to reduce the days spent waiting for computations to complete. Since the device is a shared resource, they would also like to do the initial development and verification on their laptop, then make it work on the device with the help of a software engineer or other expert. Once the work is done and the paper published, they would also like to make the software available to others who may have an entirely different device available.

Depending on the terminology of the reader's domain, what are called functions in Thalweg may also be known as filters, algorithms, kernels, mappings, or transforms. Thalweg's arrays are multidimensional arrays or matrices of up to four dimensions, and contain simple values not complex structures. Further use of the words function or array refer to these. These concepts will subsequently be clarified.

The core of this solution is a formal set of rules for designing functions that then have the potential to run on large systems. Not all algorithms can scale, but alternative algorithms solving the same problem may exist or can be developed at a later time. Thalweg has utility libraries for handling data and functions, which makes the model concrete and useable now for developing software that can take advantage of these systems. Following the API conventions allows functions to be used directly from a wide variety of scripting and high level languages once developed, which makes them available to far more users.

In isolation each of the concepts and rules in the model are not unique, but their combination creates a powerful framework that greatly simplifies the development and reusability of multi-core and distributed software.

2. Related work

The Thalweg model relies on low level languages for writing optimized functions for specific target

hardware. Both of these are used and supported by Thalweg. Each hardware platform has a set of tools developed by the vendor, and these are usually the only tools for getting the highest performance out of a device. Tools like the Intel C++ compiler can produce much more efficient programs than gcc, without any code changes. On ATI graphics cards (GPUs) Brook[1] is the native language, and for NVIDIA graphics cards CUDA[2] is used. Looking to the future, the Larrabee[3] architecture from Intel has been announced as a wide-vector multi-core architecture, which will fit into the Thalweg model and will likely have proprietary tools.

Higher level languages that target multiple platforms such as Sequoia[4], or OpenCL[5] are also methods to write functions. These languages do not support all the low level hardware features, but because they are a higher level their compilers can often be aware of global optimization and perform well in specific but not general cases in comparison to the native tools. Because of the sharing and dynamic model of Thalweg functions, it generally discourages the high-level language approach when a low level option is available.

A great deal of direct experience with developing code for CPUs, GPUs[6], and the Cell[7] was gained during working on Folding@home[8]. This more recent work enabled the Thalweg model to accommodate these new architectures in addition to the existing CPU architectures. Much of the portability and basic libraries in Thalweg are built with Cosm[9], which allows the Thalweg libraries to be completely portable in nature.

The classroom tools are partially inspired by Pintos[10], written by Ben Pfaff (blp@cs.stanford.edu). Pintos is a classroom framework for operating systems assignments, and simplifies many of the setup and grading tasks while providing students a solid starting point for projects.

3. Formal model

The Thalweg model encompasses the handling of data, functions, and error/checkpoint handling in a manner that easily allows optimized versions to be developed while maintaining portability at a high level.

3.1. Data as scalars and arrays

Thalweg has many scalar data types, including the standard 8, 16, 32, and 64-bit versions of signed and unsigned integers, along with 32 and 64-bit floating point types. These all also occur in vector forms of width 1, 2, and 4. Wider versions may also be added if

a large variety of hardware starts to use them. For portability and performance reasons the use of 64-bit types is currently discouraged, but hardware is already moving to support 64-bit values.

Thalweg arrays are abstract 1-4 dimension arrays of data elements, which can themselves be vector data types. By abstracting the data from operations on the data in this way, the main program does not have to concern itself with how the data is represented on the hardware, or even which device/host the data is currently on. Thalweg will automatically move data that is needed on different physical devices to where it is needed in the current function.

Unlike in a simple-core CPU, when programming for hundreds of cores complex data structures like trees, heaps, and graphs are not ideal. With so many cores on each chip but only one memory bus, the hardware must be able to exploit assumptions about the locality of where the next data value is in memory if all the cores are to be fully utilized[4]. Once data is spread over a distributed system, these issues become even more pronounced, and data must be more carefully arranged and localized. Many complex data structures have been successfully mapped into simple arrays for use in multi-core systems, but a comprehensive list is not given here.

3.2. Function model

Thalweg has a specific model for the way a function API and code must be written. Starting with standard reentrant functions, some additional restrictions will be added. Reentrant functions require that global variables be constant, and that any data used inside a function be provided as a input parameter. They also disallow both the use of locks for synchronization mechanisms, and the invocation of non-reentrant functions. Then the following restrictions are added:

- Every function must return an error code.
- Input parameters cannot be used for output.
- Global data must be passed as a parameter.
- If successful, all output values must be set.
- Output values may not be shared or written more than once.
- The output must be deterministic.
- Each output value must be independent.

Functions work on a barrier model, once a function is called, it will not return until all of the outputs have been calculated or an error occurs. The required error code allows the function to fail due to an internal error, or external node failures to happen,

without crashing the program. No output is considered valid until the function completes successfully, so modifying inputs would result in only partially modified outputs and corrupted inputs. This would make recovery from the error difficult or impossible without restarting at the beginning of the computation or the last checkpoint. Global data must be passed in as a parameter because in a distributed multi-core system there is no memory location that is global, only local copies on individual nodes.

The requirement that all outputs be set is a side effect of the strict division between inputs and outputs. Leaving any outputs unset and possibly uninitialized and then using them as input to another function could result in random behavior, so a value such as zero should be set so the value is not computed. Note that an output can be allocated inside of a function, so long as it is fully valid if no error occurs and is deallocated in the case of an error no violation of the model occurs.

In general, a function should do as much as possible since the cost of calling a function can be very high. In the case of a large cluster it is easy to determine why a function may take a long time to commence processing, but using a GPU on the same machine can still result in a surprisingly long call latency[6].

Outputs may not be shared, overwritten, or used as temporary storage. This would result in a race condition where the last writer wins, and locks are not allowed to avoid these conditions. The output would be less predictable for a function operating on a large amount of data, where the calculation is split onto many hosts to execute the function.

Deterministic results are required to maintain repeatability and the ability to verify results. In a distributed system full rechecking may be wanted, or two of three voting if very high assurance of correctness is needed. If random numbers are needed, then pseudorandom numbers should be generated with seed value set for that calculation. This property also makes checkpointing and process migration possible, which are desired for long running calculations.

Independence of output values is the property that allows for scaling to multi-core and distributed systems, and is the most important added property in the Thalweg model. If each output value can be calculated without knowing any other output value, then each calculation could be done on a separate core on a separate machine. This would have prohibitive overhead for a one time calculation, but for long running calculations, this is what is done, and many algorithms will scale very well in this model[4][6]. Most algorithms do need an intermediate step, or outputs mix at some point, so in these cases the overall

calculation must be broken into several steps, each of which maintaining the above properties.

It should be noted that many standard C library functions, and many functions from other libraries already come close to meeting these requirements. The most common divergence is that the returned error code and output are mixed into the return value, leaving some errors undetected. File or I/O functions violate this model, and reentrancy as well, since they read from disk or user input so lack deterministic behavior. Multi-threaded code is often already fully reentrant, but may use global variables, or data that is not from a parameter. These issues are not difficult to rectify if the inputs and outputs can be put in terms of scalars and Thalweg arrays.

3.3. Hard and soft errors

Hard errors like node crashes, power outages, and network failures are normal occurrences in multi-node systems. Since a Thalweg function may use multiple nodes to complete a computation, the evaluation of a function may fail for these reasons. Such a failure may be handled with a retry using less nodes, or may result in a fatal error, depending on the low level system being used.

Soft errors are changes to data that can be caused by radiation or excessive heat, and are random and unpredictable in nature. A white paper on error rates in modern memory subsystems by Tezzaron[11] shows that even with very low error rates modern systems which now have so many transistors and large memories, have soft errors occurring at an observable rate.

The result of these types of errors is that a Thalweg function may fail and need to be recomputed, but the overall computation can likely recover. Before distributed and multi-core systems were common a CPU could generally be counted on to always work correctly, but that is no longer the case.

4. Development framework and library

The formal model is necessary to write algorithms that can scale, but concrete tools and libraries are also required to develop, debug, and deploy applications onto real systems. Full API documentation can be found at <http://www.OpenSIMD.com/> but the key concepts will be discussed in this manuscript.

Currently the choice of hardware defines the development tools available, and a new set of tools needs to be learned for each system, with a different expertise needed for each. One of the goals of Thalweg is to make as much of the code as possible completely portable, and only require an expert for the

functions that are the bottleneck of the application. An expert software engineer can then be given the target function's reference version and tests, and will have all the tools they need to optimize that function, without requiring extensive domain knowledge.

4.1. Platforms

A platform is a specific targeted type of hardware that does not necessarily depend on the host CPU and operating system for which it was compiled. For example, SSE2 assembly is one such platform, but can be used from many operating systems using x86 hardware. CUDA is another and compiles on both Windows and Linux for NVIDIA GPUs, while OpenCL is designed to compile on many systems for multiple different GPUs.

By having the concept of a platform, individual developers can target the hardware they have, and share the results with others with the same platform, or use other platform code as a template for a new one.

4.2. Arrays

Arrays have a simple load/save model to access the data from the main program. Once data is placed in an array, Thalweg takes care of moving any arrays needed for input onto the correct hardware device before the hardware-specific version of the function is invoked. This means input data is always preloaded onto the device using it, which speeds computation.

In the situation where the inputs may be needed in more than one place, for example during different stages of an iterative calculation, moving data back and forth between devices will impose additional overhead. The goal should be to support all of the needed functions on the same device, but if this is not possible, multiple copies of constant arrays could be used. Unfortunately it is far more likely that intermediate results that need to move from device to device will be used, where data movement cannot be avoided.

4.3. Functions

The most important version of any Thalweg function is the reference version. The reference version, in standard ANSI C, is a well commented and documented version on which further optimizations can be based. If no implementation is available for the hardware at hand, then this is the version that will be used, and will still perform correctly at the expense of slower speed.

4.4. Self-tests

All Thalweg functions have a corresponding self-test routine that can determine if the algorithm and hardware are working properly and able to produce correct results. A simple device detection is done before a function is allowed to load, but this is only a detection, not a verification that the device is configured and functioning, has the right device drivers, or is fully online in the case of all the nodes in a cluster. The self-test must verify that everything is working by running a calculation with known inputs and outputs, and making sure that the results are correct. The self-test will also be used to time functions to determine which the possible platforms Thalweg should use.

Floating point calculations on different platforms may be subtly different because the hardware does rounding differently, has more/fewer bits in the internal calculations than the reference version, or may not follow IEEE math standards at all. For this reason floating point values may not match exactly in the self-tests, so for each algorithm, it must be determined how close is sufficient to be correct. This may seem minor, but is one of the most difficult issues in porting to a new platform. Is 3.14159 close enough to 3.1416 for the calculation to continue accurately? The value comparisons in the self-test need to be inexact comparisons to determine if the test passes. Integer results, however, should always agree exactly.

The self-test function needs to take long enough to trigger possible errors, and make the overhead of transferring data and invoking the function a small fraction of the test time, in order for one platform to clearly be the fastest. If the test only runs for a millisecond, Thalweg may not obtain accurate timing in order to determine which available hardware will perform the fastest on a longer run. Advanced hardware may be hundreds of times faster than the reference version, so it is best to design the self-test to run at least few seconds in the reference version.

The self-test can also be directly called to check that the device is still functioning correctly. If the device is operating incorrectly, the self-test should be able to detect this so the software can either abort or wait for the device to become stable.

4.5. Dynamic libraries

Rather than compile all functions directly into a software package, Thalweg uses dynamic libraries for all platform-specific and references functions. This allows for greater flexibility in deployed applications, since supporting new hardware just involves adding a

new dynamic library, rather than rewriting and installing an entirely new application.

Each Thalweg dynamic library contains one or more functions, along with interface and hardware detection capabilities for that platform. At runtime Thalweg provides several critical functions behind the scenes. Firstly, each loaded dynamic library detects if hardware is available matching the platform it targets. Thalweg will only load a dynamic library if it contains the correct support functions and can detect at least one device for that platform, as a way to prevent loading the wrong file. Secondly, it will not allow loading of a function from a dynamic library that does not pass the self-test. Thirdly, it determines (and the main program can remember) which of the implementations available is the fastest for that system. Combined, these things allow shipping of all the implementations, with fastest correct one being selected at runtime.

Another large advantage of this approach is that with a minimal layer of “glue” code to prepare scalars and arrays, any software that can call a dynamic library can use any Thalweg function. This includes a large number of scripting languages such as Python, Perl, and high level tools such as Mathematica or MATLAB. Many users work almost exclusively in these applications, and will never know the details of the functions or hardware. This allows shared Thalweg functions to be used by a larger audience that may not share the original choice of programming language, but would benefit from the functions.

Using dynamic libraries has the additional benefit of allowing commercial libraries to be used by any Thalweg-aware program without license incompatibilities. Such libraries may be highly optimized version of existing algorithms, or proprietary functions. This is Thalweg's way of supporting commercial vendors in the smoothest way possible.

4.6. Context

A context is an abstract collection of data and functions, from the perspective of the controlling node. A context contains both the algorithm implementations and the working data for the computation. Arrays and functions exist in one specific context, so any data is isolated unless it is directed loaded or saved from that array/context. Each context may use functions from several different platforms exploiting whatever hardware is available, but will always have at minimum the generic reference platform containing the self-tests.

For example there may be one context for a CPU and GPU combination, or four contexts one for each

of four CPUs running a single-CPU version of the algorithm. There may be one context for an SMP program running on all CPUs, or a cluster context with data spanning many hosts. In general, one context for each non-combinable computation will be needed.

4.7. Checkpointing and archives

Both in distributed systems because of node failures, and in multi-core systems because of soft errors, checkpointing has become a necessary part of any long running process. Checkpointing saves the complete state of the running calculation so that if a failure occurs only progress beyond the latest checkpoint is lost. On a calculation that could run for days a loss of a few minutes is tolerable in comparison to restarting from the beginning.

It is important before a new checkpoint is created ensure the data is still valid. Based on the calculation being performed, sanity checking of values can be performed. At minimum checking for values that are out of reasonable ranges, infinity, or NaN (Not a Number, an error that occurs in floating point calculations) should be done. The goal should be to use the nature of the algorithm to find an accurate way to check if things are still in a valid state. In a physical simulation, energies can be checked, atoms cannot be in the same place, and many other checks may be used. Unfortunately many soft errors may not trigger any of these checks if the error is small, and the error may propagate before it is noticed. Since checkpoints occur only every few minutes, pre-checkpoint verification can take some time to check for correctness before it slows down the overall calculation.

Thalweg provides a simple compressed archive format for scalar and array data. This is used to store the self-test data, and could be used for checkpointing, or preparing work to be shared with other machines for use in distributed computing. Many software packages store data in elaborate configuration files and data formats, and the code to do this parsing may be quite complex. By using the archives in the checkpoints, the initial data file parsing need only be done once. As an optimization having this parsing done at a central location the software on worker nodes can often be greatly simplified.

5. Development cycle

Implementing an algorithm in Thalweg involves three steps: design of the input and output API for the function, implementation of a self-test that can determine when the function is working correctly, and creation of a reference implementation in ANSI C. C

is used because it is almost universally known, and can produce the dynamic libraries that can be used from any higher level language. The self-test and reference version are often developed in tandem, as edge cases and possible ways for a function to fail are discovered.

These three things are required in any software, so choosing to use Thalweg imposes no additional development work on the researcher, but adds significant benefit and flexibility versus targeting only one platform.

The user is not required to write any non-portable code (code that deals with platforms, hardware detection, or loading data onto a specific device). Optimized implementations of a function will be aware of the hardware, and can find and exploit every performance method available, but that the designer of those functions need not be concerned with these issues so long as they follow the model correctly.

6. Optimization of functions

Once the reference code is written and proves to solve the problem correctly with a robust self-test, the focus can turn to optimization. High-end hardware requires a niche set of skills to get the maximum performance from the device. These skills can span compilers, chip architectures, memory performance characteristics, networking, high speed interconnects, and a variety of software frameworks for handling communications between nodes. This requires specialized software engineering skills which scientific users are unlikely to possess.

In the other direction, the software engineer who is working on a fast implementation should not need to know about anything about the detailed algorithms used beyond that function. This removes the need to find people who are simultaneously software engineers and experts in the application domain. In Folding@home, the ATI, NVIDIA, and PS3 versions are all now written and maintained by software engineers at those companies, not at Stanford. This arrangement has worked well, as the focus of the researchers can be completely on the science, while the software engineers at the companies can manage architectural details and focus on performance.

The effort of learning the native tools and writing an optimized version of a function (or hiring someone to do it) requires justification in a climate where generic languages are available and losing a factor of two in performance is considered acceptable in academic circles. The easiest justification is that an individual researcher is likely to have access to only one type of cluster, so if learning one hardware language is necessary, it should be the one matching

the local hardware. This argument is weakened when as there is a need to share code with others on another platform, but if they are provided a reference version and they are familiar with their own hardware, this is not a significant barrier to sharing.

Once hundreds of servers are assembled in a room and the networking and cooling is paid for, an increase in performance of even 10% at the cost of a few weeks of effort by a software engineer will quickly return in saved electricity, efficient resource utilization, and decreased waiting times. In addition to the time and money, the electricity used by non-optimized code also has environmental and climate implications. Improvements when the low level language for the specific hardware is used is often far more than 10%, sometimes multiples of the generic performance. The Thalweg model of building a reference and then optimized version of functions is consistent with these observations and goals.

7. Sharing functions with others

Writing code is almost never the end goal of research, it is at best a necessary evil required to implement the algorithms needed. As practical, any functions created should be made available when the related papers are published. The Thalweg website OpenSIMD.com will host a library of both reference and accelerated versions of algorithms, along with their self-tests and documentation. The library will allow sharing of code and expertise, saving time and effort while adding citations for the creators.

8. Thalweg in the classroom

Programming curriculum in high schools and colleges does not currently encompass many if any multi-core or distributed computing concepts. Threading is being taught, but in the context of web servers and other throughput based architectures. The focus there is on getting lots of things done, not getting one large thing done quickly. Coordinating multiple cores and using distributed systems are considered advanced or graduate study.

This needs to change as multi-core systems becomes ubiquitous, and that means simple ways must be found to teach these concepts. Most people encounter vector and data-parallel models in everyday activities, so the prevailing consensus that this type of programming is foreign is unfounded.

In any project-based course, regardless if distributed and multi-core concepts are the focus, fitting all the required material and projects into a single quarter is difficult. Often students have to learn

a large number of unrelated technologies in order to finish the a project. Setting up the project, compiler, and build tools, and often a separate OS in a VM as, adds time and frustration to the student's experience. By isolating the pure functions and data from the rest of the system, Thalweg allows students to work only on the methods or techniques that are to be taught, and the rest of the frameworks can be provided. Grading is greatly simplified by the production of dynamic libraries, the library built by the student run through a battery of tests (which is just a more elaborate version of the self-test) designed by the instructor or teaching assistants to check for proper outputs given test inputs.

There does seem to a be a higher barrier to learning these concepts if the perspective programmer has already learned single-core and multi-threading methods. A similar effect was seen during the transition to object oriented languages such as C++. Originally these were seen as hard concepts that students should be isolated from, but now initial programming courses are often taught using Java and utilize objects extensively.

9. Conclusions and future work

The days of simple, single-core, standalone architectures are now in the past. Thalweg provides a framework for writing shareable functions in a way that they can be accelerated on current and future hardware.

The Thalweg site OpenSIMD.com currently hosts forums where users can discuss functions, platform issues, and optimization, and will eventually provide a place to share Thalweg functions. This should help with the goal of reducing the programming burden on researchers, and reduce the time spent waiting for computations to finish.

Over time the collection of algorithms that are available should grow to include many commonly used and important algorithms.

We thank NSF and NIH for their support of this work, and the many people that have provided feedback on the Thalweg model and APIs.

10. References

- [1] Ian Buck, et al., "Brook for GPUs: Stream Computing on Graphics Hardware," ACM Transactions on Graphics (TOG), vol. 23, (3), pp. 777-786, August 2004.
- [2] "OpenCL - The open standard for parallel programming of heterogeneous systems". 2008. [Online]. Available: http://www.nvidia.com/object/cuda_home.html.
- [3] Larry Seiler et al., "Larrabee: a many-core x86 architecture for visual computing", ACM Transactions on Graphics, vol. 27, (3), pp. 1-15, August 2008.

- [4] K. Fatahalian, et al., "Sequoia: Programming the Memory Hierarchy", Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006.
- [5] OpenCL website. 2008. [Online]. Available: <http://www.khronos.org/opencl/>.
- [6] M. S. Friedrichs, et al., "Accelerating Molecular Dynamic Simulation on Graphics Processing Units," Journal of Computational Chemistry, 2009.
- [7] E. Luttmann, et al., "Accelerating molecular dynamic simulation on the cell processor and Playstation 3," Journal of Computational Chemistry, vol. 30, (2), pp. 268-274. July 2008.
- [8] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, V. S. Pande, "Folding@home: Lessons From Eight Years of Volunteer Distributed Computing," Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), May 2009.
- [9] The Mithral website (Cosm). 1995. [Online]. Available: <http://www.mithral.com/>
- [10] Ben Pfaff, "Pintos," [Online]. Available: <http://pintos.benpfaff.org/>.
- [11] "Soft Errors in Electronic Memory - A White Paper," white paper, Tezzaron Semiconductor, Jan. 2004.