Transitioning Data Flow-Based Visualization Software to Multi-Core Hybrid Parallelism

David Camp
Lawrence Berkeley Natl. Laboratory
Berkeley, CA, USA
Email: dcamp@lbl.gov

E. Wes Bethel
Lawrence Berkeley Natl. Laboratory
Berkeley, CA, USA
Email: ewbethel@lbl.gov

Hank Childs
Lawrence Berkeley Natl. Laboratory &
The University of Oregon
Email: hchilds@lbl.gov, hank@uoregon.edu

***Abstract*—Many of today's programs for supercomputers are designed for distributed-memory parallelism, but not for shared-memory parallelism. As architectural trends for supercomputers suggest an ever-increasing number of cores per node, these projects must evaluate whether they can benefit from hybrid parallelism — parallelism that blends distributed- and shared-memory approaches — and whether the costs for migrating to new architectures are prohibitive. With this research effort, we explore whether a data flow-based visualization tool could be easily migrated to a hybrid parallel setting, what the challenges are, and what benefits hybrid parallelism achieves. For results, we find that only a relatively small investment was required to transition the tool and it significantly improves performance and also reduces memory usage and communication costs.**

I. Introduction

In the 1990s, supercomputer architectures transitioned from an era of vector processing to an era of clusters made up of nodes of commodity hardware. At the beginning of this transition, it was common for each node of the cluster to have only a single CPU on each node. Over the last decade, however, computing centers have incorporated multi-core nodes into their designs: dual-core nodes, then quad-core nodes, and now frequently a dozen or more cores per node.

Unfortunately, many large software packages were designed to handle the parallelism challenges from distributed-memory nodes, but not the parallelism challenges of shared-memory cores within a node. This is because the transition from vector processing to commodity clusters required a significant investment in designing software for a completely different architecture. Many of the resulting efforts (reasonably) focused on distributed-memory parallelism; since nodes had only a single core at the time, distributed-memory parallelism was the pressing challenge. By and large, these efforts created a Processing Element (PE) — an instance of their program — on each node and used the Message Passing Interface (MPI [1]) for execution control, synchronization, and interprocessor communication. On platforms comprised of multi-core processors, many of these MPI-based tools simply sidestepped the issue by running a PE on each core, in effect treating each core as if were a node. In effect, the programs maintained a distributed-memory approach, assigning a PE to each core on a node and forcing cores within a node to communicate with each other using message passing.

Hybrid parallelism refers to the practice of using a combination of shared- and distributed-memory techniques, namely using distributed-memory techniques across nodes, and shared-memory techniques within a node. The technique has the ability to improve performance and lessen resources consumed, specifically I/O, network communications, and memory. Further, from the perspective of an individual core, these resources are diminished as the number of cores increase; the algorithms we employ must be responsive to this. The benefits of hybrid parallelism, and the best way to implement the approach, has recently been an active area of research.

With this short paper, we consider a specific instance of a program designed for distributed-memory parallelism that has survived into the hybrid parallel era. The program is data flow-based and is used to visualize and analyze massive data sets. The developers of this program — as well as the developers of many other programs which also were designed only for distributed-memory parallelism — are faced with a key decision: Should they adapt their existing program? Should they start over from scratch? Or should they do nothing? We attempt to inform all three questions by exploring the approach of modernizing an existing program to deal with multi-core nodes, including the costs and benefits of the approach. The costs inform the tradeoff between modernizing and re-writing, while the benefits inform the question of whether action is required.

The contributions of this short paper are aimed at answering questions pertinent to developers of distributed-memory only data flow programs. Specifically:

- **Cost:** How much effort is required to transition data flow programs to a hybrid parallel setting? Are there special advantages or disadvantages unique to data flow? What are the best approaches?
- **Benefit:** What benefits does hybrid parallelism add to data flow programs? (I.e., is "do nothing" a viable option?)

In our study, we find that, because of the data flow nature of the program, only (relatively) small changes were required to enable hybrid parallelism in an existing program. We also demonstrate significant benefits for performance, memory, and communication.

II. Related Work

A. Hybrid Parallelism

As already discussed, hybrid parallelism combines distributed- and shared-memory techniques. For distributed-

memory parallelism, nearly all applications use the MPI [1] to manage coordination between its PEs. For shared-memory parallelism, there are more options, including POSIX threads [2], OpenMP [3], and Intel Thread Building Blocks [4].

Early hybrid parallelism work focused on benchmarking well-known computational kernels [5], [6]. Subsequent work explored specific impacts of hybrid parallelism for visualization [7]. Specifically, Howison et al. showed a factor of two speedup and significant memory savings when volume rendering using $216,000$ cores with weak scaling [8] and strong scaling [9]. Camp et al. studied performance benefits for streamlines [10] and demonstrated order-of-magnitude speedups.

B. Overview of VisIt

VisIt [11] is an end-user tool for visualizing massive data sets. Its development started in the year 2000 — when hybrid parallelism was not frequently considered by developers — and continues to be heavily used and actively developed today. The project has been downloaded over 200,000 times and is used around the world, including many of the supercomputers on the Top500 [12]. VisIt contains over 1.5 million lines of C++ code, meaning reimplementing the code base from scratch for hybrid parallelism would be a substantial undertaking.

VisIt follows a client-server model, where the client runs on the user's desktop computer, and the server runs where the data can be accessed and processed, likely on a remote supercomputer. The server is parallelized; data is loaded in parallel, processed in parallel, and rendered in parallel. VisIt has scaled to tens of thousands of cores and been used to look at data sets with trillions of cells [13].

C. Data Flow In VisIt

VisIt's server is based on data flow techniques. Visualization algorithms are encapsulated as filters, which are combined into data flow networks referred to as pipelines. VisIt contains over 300 filters, each of which performs a different data manipulation, transformation, derivation, or rendering. Users themselves decide which filters to place into a pipeline, dynamically creating exactly the visualization they want to see.

The flow is demand-driven; filters go through an "update-execute" cycle. The cycle begins when the pipeline's sinks are asked to "update." The update requests propagate up the pipeline until they reach sources. After the sources produce data — usually by reading it from a file — data flows down the pipeline and filters execute on the data as it comes. VisIt's design is similar to (and inspired by) that of the Visualization ToolKit (VTK) [14], but contains key enhancements. Specifically, a contract-based mechanism [15] enables filters to communicate with each other both their requirements and their opportunities for optimizations, for example data culling, where only the necessary regions of the data are loaded and processed.

III. Approach

VisIt's filters are implemented as C++ classes, with a deep inheritance hierarchy. The base class for all filters is `avtFilter`, which is an abstract type. The middle portions of the inheritance hierarchy specialize what types of data a filter operates on, e.g., mesh-based data, images, etc., or specialize its processing strategy. Concrete types implement a specific algorithm and it is these concrete types that are instantiated as part of a data flow network. This design is common to many data flow libraries; the management of execution is the same across the filters, so it is abstracted into a base class that many derived types can share.

VisIt uses data-parallel techniques to achieve parallelism. Large data sets are divided into pieces (corresponding to spatial regions); VisIt partitions these pieces over its PEs and each PE operates on its own pieces. Typically, there are many more pieces than PEs, since the simulation code often divides the data set into pieces itself for its own parallelism, and since the simulation frequently has many more PEs than the visualization program. The majority of the algorithms in VisIt are embarrassingly parallel, meaning that no coordination is required between the PEs when parallelizing. In this case, each PE operates on its pieces and the results are aggregated later on in the execution, typically during rendering.

Our goal was to have one PE on each node, and to have all the cores share that PE's workload. Previously, each PE would iterate over the pieces assigned to it (one at a time), so having each core operate on a piece was a natural extension to the existing scheme. Further, since a hybrid parallel approach reduces the number of PEs, the number of pieces per PE would go up proportionally, guaranteeing that there would be sufficient pieces to assign to the cores. Our question became: how should we schedule work to cores in a way that could remain coordinated?

Our scheme was to, for each filter execution, acquire threads from a thread pool, complete the execution, and then return the threads to the thread pool. We considered an alternate scheme where each thread would be assigned a piece and execute the entire pipeline on that piece before getting more work. We did not choose this scheme since, although many filters are embarrassingly parallel, those that require parallel coordination would require significant extensions in this context. Further, our selected scheme imposes fewer restrictions on how to carry out shared-memory parallelism. We were able to provide an implementation in the base class that worked for approximately 95% of the filters.

For the filters that require special coordination in parallel — i.e., the remaining 5% — there are two options. The first option is to disable hybrid parallelism for that filter. In this case, only one core is used from a node, which is inferior to the distributed-memory only case. However, it allows the program to continue, it takes no additional developer effort, and the rest of the program can still run in a hybrid parallel manner. The second option is to not use the infrastructure from the base class and, instead, implement custom shared memory parallelism in a way that optimizes filter execution. For example, the streamline work referenced in [10] requires sophisticated queuing of shared-memory work that goes beyond our simpler scheme. Since this scheme was already implemented in VisIt (on an experimental branch), it can simply replace the basic threading scheme with its own existing scheme.

Our work items to realize this vision are listed below. Time spent to achieve each work item is listed parenthetically.

- **Threading support (3 weeks):** We added an abstraction for an execution manager, which carries out assignment of work to resources. Within the execution manager, we implemented support using using POSIX threads (pthreads), including management of a thread pool, thread creation and deletion, and thread joining.
- **Filter-level support (5 weeks):** We added code in the base classes to engage the execution manager to round-robin pieces over the threads. Since most filter implementations had already abstracted away parallelism and were simply focused on how to execute on a given piece, 95% of filters could use this infrastructure and did not even need to be aware that they were now running in a hybrid parallel mode.
- **Auditing of derived types (5 weeks):** Although the thread support could be added in the base class, some filters were implemented in a non-thread-safe way. The most common example would be a filter's method to manipulate a data set modified data members associated with the instance of the class. To catch these cases, we used VisIt's regression test suite, which contains over $7,000$ tests.
- **Auxiliary infrastructure (3 weeks):** VisIt's timing infrastructure outputs one file per PE and had to be modified to deal with multiple cores creating timing events simultaneously (through semaphores). A similar issue arose with the creation of log files. Finally, VisIt's job launching had to be modified to reflect hybrid parallel job submissions.

In total, this effort was approximately four months of work. The total development time in VisIt since its beginning is approximately 100 person-years, making the hybrid parallelization effort relatively quick. We note, however, that although we have transitioned the infrastructure of VisIt to hybrid parallelism, we have not completely finished the transition, especially with respect to migrating non-embarrassingly parallel filters.

IV. Measuring the Benefits of Hybrid Parallelism

A. Study Overview

We compared the results of running a distributed-memory version of the program with a hybrid parallel version. The test was designed to isolate the difference between the two parallelism approaches. Both configurations used identical hardware, worked on the same data sets, and applied the same visualization algorithms.

We ran the tests on the Hopper machine at Lawrence Berkeley National Laboratory's NERSC facility. Hopper's compute nodes each contain 32GB of memory and two twelve-core AMD 'MagnyCours' 2.1-GHz processors, for a total of twenty-four cores per node.

The experiments loaded output from a GenASiS [16] simulation of the magnetic field surrounding a solar core collapse, calculated an isosurface, and rendered the resulting surface (see Figure 1). The data set consisted of over two billion cells, broken into 512 pieces.

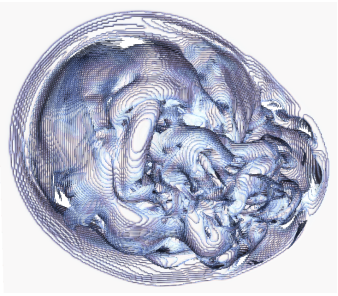

Fig. 1. Rendering of an isosurface of a solar core collapse. Identical images were rendered by a distributed-memory version of VisIt, and by a hybrid parallel version, and identical hardware was used for both configurations. The differences in performance and resource consumption inform the benefits of hybrid parallelism.

We ran the experiments on three nodes. For the distributed-memory tests, we ran with 72 PEs. For the hybrid parallel tests, we ran with 3 PEs, and each PE used twenty-four cores.

B. Results

We compared the performance, memory usage, and communication patterns of the tests.

With respect to performance, we found that the hybrid parallel version was consistently faster. This is because, with isosurfacing, the amount of work per piece varies based on how much of the data set intersects the isosurface. The hybrid parallel version effectively balanced this load dynamically; the distributed-memory version had some PEs sitting idle while other cores on the same node still had work to perform. This imbalance is crucial, since the results are not displayed until all PEs have finished. The individual times for each PE can be seen in Figure 2. Over five tests, the hybrid parallel version took between 8.28s and 8.45s, while the distributed-memory version took between 13.5s and 14.9s. The average speedup was 1.67X.

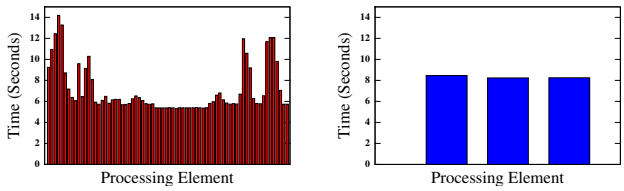

Fig. 2. The left figure shows the time for the distributed-memory approach and its 72 PEs to process its pieces. The right figure shows the time for the hybrid parallel approach and its 3 PEs to process its pieces. The hybrid parallel version averaged being 1.67X faster.

The only aspect of our test that emphasized communication was the rendering. Parallel rendering is a two-step process [17]. First, each PE renders its own surface data into a sub-image. Then the PEs exchange the sub-images and composite them together using information about the depth of the surface. With the distributed-memory test, 72 sub-images are exchanged, while, with the hybrid parallel version, only 3 sub-images are exchanged. As a result, the compositing times for the hybrid parallel version are faster, averaging 0.3s versus 0.5s for the distributed-memory version. That said, we note that we have not yet converted our surface renderer to hybrid parallelism. As a result, only one core performed the first step (rendering of surface data), and the total rendering time was overall slower with the hybrid parallel version. Hybrid parallelism will be faster after the conversion, because of its advantages in compositing time.

The hybrid parallel version uses less memory, because it only loads one version of the binary. At the end of execution, the distributed-memory version uses 22.3GB, while the hybrid parallel version uses 20.7GB, for a saving of 1.6GB. This is because each instance of the program takes almost 70MB. Realistically, 1.6GB out of the available 32GB is noteworthy, but not significant. However, since architectural trends have more and more cores placed on a node and memory staying constant, this savings will become more and more important.

V. Conclusion and Future Work

We found that we only needed to modify relatively few locations to convert a large application to hybrid parallelism. This is because the data flow pattern lends itself to an inheritance design and the hybrid parallelism can be addressed in a base class of an inheritance hierarchy and benefit many derived types. Further, the large majority of visualization data flow algorithms are embarrassingly parallel. If this was not the case — i.e., if many of the algorithms required parallel coordination — then this effort would have been a much more significant undertaking.

The scope for this effort was for multi-core CPUs, not many-core GPUs. Multi-core CPU platforms lend themselves to the type of refactoring described in this paper, since multi-core programming environments are the same as those of single core. (In our case, VisIt's C++ code is extended to include POSIX threads.) Many-core GPU platforms require languages that enable fine-grained parallelism, such as CUDA [18] or OpenCL [19]. Three visualization libraries have emerged that tackle fine-grained parallelism: PISTON [20], DAX [21], and EAVL [22]. Incorporating one these libraries into a large tool like VisIt — and thus making it many-core capable — is a feasible task, especially since our newly added execution manager was designed with this use case in mind. However, the many-core transition will be significantly more time-consuming than the one considered in this study: the visualization routines in VisIt make up almost $700,000$ lines of code and this code would have to be reimplemented (and possibly re-thought) for a many-core setting.

Acknowledgments

Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI – The Complete Reference: The MPI Core, 2nd edition*. Cambridge, MA, USA: MIT Press, 1998.

[2] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[4] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor*. O'Reilly Media Inc., 2007.

[5] G. Hager, G. Jost, and R. Rabenseifner, "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes," in *Proceedings of Cray User Group Conference*, 2009.

[6] D. Mallón, G. Taboada, C. Teijeiro, J. Tourino, B. Fraguela, A. Gómez, R. Doallo, and J. Mourino, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *16th European PVM/MPI Users' Group Meeting, (EuroPVM/MPI'09)*, September 2009.

[7] E. W. Bethel, D. Camp, H. Childs, C. Garth, M. Howison, K. I. Joy, and D. Pugmire, "Hybrid Parallelism," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, Oct. 2012, pp. 261–290.

[8] M. Howison, E. W. Bethel, and H. Childs, "MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Apr. 2010, pp. 1–10.

[9] ——, "Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 18, no. 1, pp. 17–29, Jan. 2012.

[10] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy, "Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 17, pp. 1702–1713, Nov. 2011.

[11] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, Oct. 2012, pp. 357–372.

[12] Top500, "Top 500 supercomputer sites," http://www.top500.org/, 2013.

[13] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. Weber, and E. W. Bethel, "Extreme Scaling of Production Visualization Software on Diverse Architectures," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 22–31, May/Jun. 2010.

[14] W. J. Schroeder, K. M. Martin, and W. E. Lorensen, "The design and implementation of an object-oriented toolkit for 3d graphics and visualization," in *VIS '96: Proceedings of the 7th conference on Visualization '96*. IEEE Computer Society Press, 1996, pp. 93–ff.

[15] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, and B. J. W. A. D. N. Max, "A Contract-Based System for Large Data Visualization," in *Proceedings of IEEE Visualization (Vis05)*, Oct. 2005, pp. 190–198.

[16] E. Endeve, C. Y. Cardall, R. D. Budiardja, and A. Mezzacappa, "Generation of Strong Magnetic Fields in Axisymmetry by the Stationary Accretion Shock Instability," *ArXiv e-prints*, Nov. 2008.

[17] C. Hansen, E. W. Bethel, T. Ize, and C. Brownlee, "Rendering," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, ser. Chapman & Hall, CRC Computational Science, E. W. Bethel, H. Childs, and C. Hansen, Eds. Boca Raton, FL, USA: CRC Press/Francis–Taylor Group, Nov. 2012, pp. 49–60, http://www.crcpress.com/product/isbn/9781439875728, LBNL-6324E.

[18] NVIDIA Corporation, *NVIDIA CUDA^{TM} Programming Guide Version 3.0*, http://developer.nvidia.com/object/cuda_3_0_downloads.html, 2010.

[19] Khronos Group, "OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems," http://www.khronos.org/opencl/, 2011.

[20] L.-t. Lo, C. Sewell, and J. Ahrens, "PISTON: A portable cross-platform framework for data-parallel visualization operators." Eurographics Symposium on Parallel Graphics and Visualization, 2012, pp. 11–20.

[21] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma, "Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2011, pp. 97–104.

[22] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros, "EAVL: the extreme-scale analysis and visualization library," in *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012, pp. 21–30.