LA-UR-10-02239

*Approved for public release;*
*distribution is unlimited.*

| | |
|---|---|
| *Title:* | Quantitatively driven visualization and analysis on emerging architectures |
| *Author(s):* | Patrick S. McCormick 107793 CCS-1, James P. Ahrens 113788 CCS-1, Erik Anderson University of Utah, Mathew E. Maltrud 094509 T-3, Steven L. Martin University of Utah, Carson Brownlee University of Utah, Jeff Thornton Inman 144219 HPC-1, Mark Kim University of Utah, Lee Nau University of Utah |
| *Intended for:* | Journal of Physics: Conference Series - Scientific Discovery Through Advanced Computing, Volume 125, 2008 |

Journal of Physics: Conference Series **125** (2008) 012095 doi:10.1088/1742-6596/125/1/012095

# Quantitatively driven visualization and analysis on emerging architectures

**Patrick McCormick, Erik Anderson, Steven Martin, Carson Brownlee, Jeff Inman, Mathew Maltrud, Mark Kim, James Ahrens and Lee Nau**

Los Alamos National Laboratory, Los Alamos, NM 87545, USA

E-mail: pat@lanl.gov

**Abstract.** We live in a world of ever-increasing amounts of information that is not only dynamically changing but also dramatically changing in complexity. This trend of "information overload" has quickly overwhelmed our capabilities to explore, hypothesize, and thus fully interpret the underlying details in these data. To further complicate matters, the computer architectures that have traditionally provided improved performance are undergoing a revolutionary change as manufacturers transition to building multi- and many-core processors. While these trends have the potential to lead to new scientific breakthroughs via simulation and modeling, they will do so in a disruptive manner, potentially placing a significant strain on software development activities including the overall data analysis process. In this paper we explore an approach that exploits these emerging architectures to provide an integrated environment for high-performance data analysis and visualization.

## 1. Introduction

Simulation and modeling provide a cornerstone for many efforts within the broad scientific community. The hardware requirements for these applications range from a single desktop computer to machines with thousands of processors. Over the past ten years the increasing clock speeds of microprocessors has provided the performance improvements required to meet the ever-increasing demands of computational science. The fundamental processor design supporting these tasks is undergoing a revolutionary change that will impact systems at all scales. Furthermore, datasets are continuing to increase in both resolution and complexity. These factors place a significant challenge to successfully leveraging the power of future large-scale computing resources. This section provides an overview of the current trends in microprocessor design, the growth of data sizes and complexity, and discusses the impact they will have on the overall data analysis and visualization process.

While improvements in semiconductor manufacturing continue to allow for increasing transistor densities, the physical limits of these designs (heat dissipation and power consumption) are placing a significant strain on supporting faster clock speeds. In response, the computer industry has switched to building chips assembled from several simplified, more power efficient processor cores. With this change, future performance improvements will be achieved not by increasing clock frequencies but by increasing the amount of parallelism within processors. Over the next several years this trend will lead to fundamental changes in the way we design algorithms and develop software.

Coupled with the changes in the design of central processing units (CPUs), graphics hardware has rapidly evolved over the past decade from a fixed-function hardware pipeline, into a programmable parallel processor with hundreds of cores. These highly parallel processors have computational and

 

memory bandwidth rates that significantly exceed the capabilities of the most recent CPU designs. For example, AMD and NVIDIA have recently announced new graphics processors (GPUs) with approximately one teraflop of (theoretical) single-precision peak performance [1, 2]. (Note that while both processors support double-precision operations, they are expected to perform at significantly lower rates on the order of 100 gigaflops.) Furthermore, several efforts have shown that GPUs can provide substantial performance improvements and can be successfully used in cluster-based computing environments [3, 4, 5]. Although there are distinct disadvantages and limitations of using GPUs for general-purpose computation, manufacturers are slowly introducing hardware features and programming tools that are making them more suitable for the scientific community. Finally, the improvements in graphics processors are also supporting the development of new interactive rendering and visualization techniques that were difficult, or impossible, on previous generations of the hardware. Much like the changes to CPU architectures, the substantial levels of parallelism, and the nuances of programming models and hardware designs used in GPUs represent a significant challenge to developing efficient, high-performance software.

Further trends within the microprocessor industry suggest that it is likely that many features of GPUs will be incorporated as additional cores within the traditional homogeneous core designs used today. IBM's Cell Broadband Engine provides an early glimpse into a chip that leverages this heterogeneous core design [6]. While such designs have the potential to improve performance, it is likely to further complicate programming models. The extent to which software can fully leverage increasing levels of parallelism within a single chip, in the form of homogeneous or heterogeneous cores, is likely to be the limiting factor on overall performance in the future. This revolution in processor design will have a broad impact across all of modeling and simulation – including data analysis and visualization operations. This will become critical as we confront the continuing challenge of ever-increasing data set size and complexity.

Over the past decade we have seen the sizes of datasets increase along with the growth in processing power. This situation has quickly overwhelmed our capabilities to explore, hypothesize, and thus fully interpret the underlying details within these datasets. In addition to increasing spatial resolution, we are now producing data with ever more complexity. This complexity is due to increasing numbers of variables that are required to faithfully represent the behavior of complex and dynamic systems and advanced techniques such as adaptive grid structures. As computational resources begin to reach peta- and exa-scale sizes, we will face a significant challenge in addressing the needs of analyzing and visualizing these data. There are two fundamental issues that arise when processing datasets of this magnitude. First, efficiently and effectively processing enormous datasets will require computational capabilities and capacities similar to those that were used to produce the data. It is simply the case that we are well on the way to producing more data than we can possibly store. Second, the set of algorithms and supporting code used to comprehend the myriads of information contained within massive datasets will also increase in complexity. Each of these aspects will be directly influenced by the architectural issues discussed above. The challenge of developing software that scales with the number of processor cores will directly impact our ability to deal with increasing amounts of information and thus limit our ability to make new scientific discoveries.

## 2. Impact on the scientific workflow

The performance growth of computers has allowed them to become a fundamental part of scientific research. This has allowed us to replace or augment the traditional approaches used in the pursuit of understanding complex systems ranging from the interactions of subatomic particles, to the large-scale phenomena responsible for the formation of the universe. In this paper we explore a subset of the data analysis challenges faced by applications as they confront rapidly changing computer architectures and the continuing onslaught of data. Of particular importance to our discussion is the relationship between the overall data analysis process and the scientific method. Figure 1 presents a simplified version of a workflow for the associated set of tasks. The stages of this process consist of an iterative series

of both qualitative and quantitative studies of data. The qualitative presentations assist in the overall understanding of the data. In comparison, a combination of numerical studies and quantitatively driven visualization operations contribute to the interpretation and assessment of the results. Together these two sets of tasks provide support by gathering observable, empirical and measurable data to assist in the formulation and testing of hypotheses. Throughout this process visualization plays a critical role in helping scientists understand large amounts of information. As the size and complexity of datasets grow, it becomes increasingly difficult to display information at interactive rates. This situation affects the ability to leverage the full power of our visual system; thus directly impacting the efficiency of the workflow.

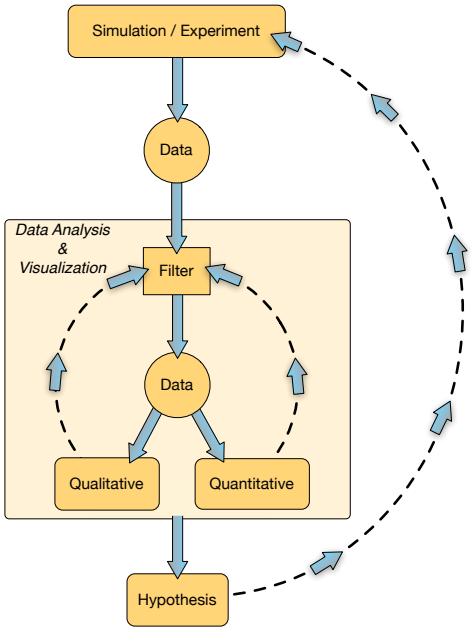


**Figure 1.** Workflow for data analysis and visualization relative to the overall scientific method. The results from a simulation, or experiment, are run through a series of filters that are used to produce both qualitative and quantitative results. This process is repeated until a hypothesis is tested and/or found to be true with further experimentation.

A wide variety of tools is available to scientists to use during this workflow. These tools range from commercial and open-source products to custom application-centric software. It is not uncommon to find scientists within the same fields using different software to accomplish the same tasks. This set of tools is often very disjoint in terms of both the features they provide as well as their abilities to leverage high-performance computing resources and efficiently process large datasets. For example, while many simulations require hundreds to thousands of processors, the resulting data is often processed using tools that are capable of running only on a single desktop computer. In the worst case, this lack of cohesion can significantly impact productivity and often leaves vast portions of data unexplored. The difficulty of developing analysis software that can process extremely large datasets, leverage the power of large-scale computing resources, and meet the specific needs of individual application domains presents an significant roadblock to the scientific community. We believe a portion of these challenges can be addressed by providing users with direct programmatic control over fundamental numerical and visualization operations.

## 3. Direct programming

Many of our past efforts have focused on providing interactive visualization and rendering algorithms for dealing with large-scale data sets. Not surprising, such approaches have solved only a very small portion of the overall challenge faced by today's scientific applications. Although they represent possible building blocks, these algorithms fail to fully address the higher-level issues related to providing users with a set of flexible tools for exploring and evaluating data in both a qualitative and quantitative fashion. As the popularity of several software packages has shown, it is beneficial to have the ability to directly perform mathematical operations when exploring and analyzing data. For example, the success of the Python programming language combined with libraries like `matplotlib` speaks for itself [7, 8]. However, these tools rarely address the full set of challenges associated with scaling to process large datasets and only a few are beginning to leverage the full capabilities of emerging hardware. Addressing this challenge has been the fundamental focus of the Scout project, which provides a parallel programming language for data analysis and visualization [9, 10]. The use of an explicit parallel language helps to hide the nuances of the programming models of the underlying architectures – including multicore CPUs, GPUs, and cluster-based environments. Furthermore, analysis, visualization, and rendering computations can all be carried out within the language (and supporting runtime environment) and potentially improve the productivity of the workflow discussed in Section 2. As we have previously presented, we see this approach as playing a fundamental role in an integrated data analysis and visualization framework aimed at assisting in the generation of new scientific insights and results [11].

Before presenting an advanced set of operations using Scout, we begin with two simple examples that present the basic functionality and provide an introduction to the programming model. Scout is based on the ideas and structures of the C* and CM FORTRAN programming languages [12, 13]. Both of these languages, initially designed for Thinking Machines supercomputers, support a data-parallel programming paradigm that fits well with other languages used within the community and can be matched to the capabilities of the underlying hardware. Scout extends these languages by adding structures that support visualization and rendering operations that support controlling the color generated for each resulting pixel. All of the examples presented in this paper present language keywords in red and built-in language functions as blue.

```
// Simple 2D color-mapped slice example.
// pt = potential ocean temperature.

mask(pt,-999); // mask out land values

render with(shapeof(pt[:,:,0:0]) {
  // use hue-saturation-value space
  // to color blue to red (cool to warm)
  image = hsv(240*(1.0-norm(pt)),1,1);
}
```

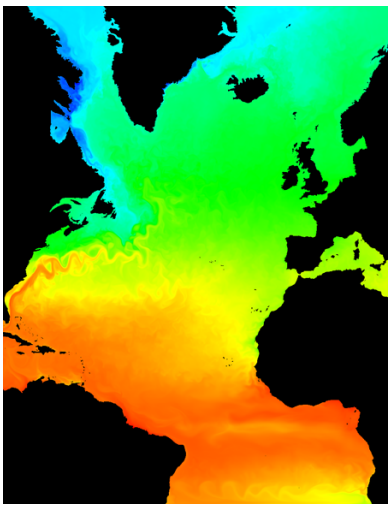

**Figure 2.** Sample Scout code showing a simple two-dimensional visualization operation. The `mask` operator removes locations with a value of $-999$ from any computations.

The first example shown in figure 2 shows how a two-dimensional slice can be extracted from a three-dimensional field, and then rendered according to the minimum and maximum data values within

the slice. The `with` construct can be best thought of as enabling a set of processors that execute the operations contained within the enclosing braces in parallel. In this example a range operator (`[:,:,0]`) is used to select all the values over the $x$ and $y$ dimensions and only the first slice of the $z$ dimension of the supplied dataset – producing a two-dimensional array of data values. The body of the supporting `with` statement then simply normalizes the input values into the range $[0.0,\ldots,1.0]$ and colors each resulting pixel in the output `image` by mapping them into the $[0,\ldots,240°]$ hue range (blue to red) of the HSV color space [14].

In a similar fashion, we can compute an isosurface and color it by another field. Figure 3 shows the code for and results of executing this operation on the same dataset presented in figure 2. Like the previous example, this code also enables a set of parallel processors but these are only active over the generated locations within the isosurface; therefore, the enclosed `range` operation is only for the salinity values over the surface and not a global data range. During this process, Scout will leverage a combination of both CPU and GPU resources to complete the task; this is done regardless of the computational platform being used, be it a laptop computer or cluster of systems. The next section provides further details on Scout's ability to leverage the power of multicore and cluster-based computing resources.

```
// Isosurface at 20.0 degrees.
render isosurface with(temp, 20.0) {
  // Find the range of salt values over
  // the generated surface.
  float smin, smax, srange;
  range(salt, smin, smax);
  srange = smax - smin;
  // Color output pixels by salt range
  // (blue to red).
  image = hsv(240 * ((salt - smin)/srange),
              1, 1);
}
```

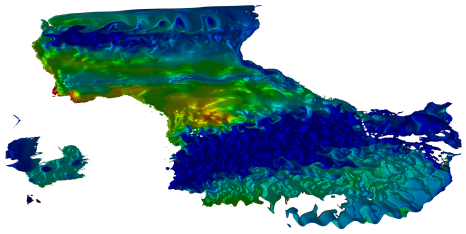

**Figure 3.** Sample Scout code showing a simple isosurface of temperature at 20 degrees Celsius colored by corresponding salinity values. Note that the inclosing `with` block forces computations to occur only over the points within the generated surface.

## 4. Examples

The examples presented in the previous sections used fundamental parts of Scout's basic operations that were originally designed to leverage the power of the GPU within a single desktop computer. In this section we present two detailed examples exploiting new features that are currently being developed. The first example presents our findings for leveraging the power of a cluster system built from multicore CPUs and GPU processors.

### *4.1. Leveraging Multi-core and Cluster Resources*

Although they have received very little attention within the research community, conventional two-dimensional visualization operations comprise a significant majority of the techniques used by the scientific community to reveal structure and the relationships between multiple variables. These techniques serve as an important link between computational results and traditional methods of understanding observed values. Unfortunately, many of the software packages used to produce these plots do not scale to support large data or high-performance computing resources. Thus many users are left running batch processes, or significantly down sampling data, to produce these simple results. In

this section we present the results of using Scout's underlying runtime system to support an interactive, programmable scatter plot that can run on systems ranging from laptops to visualization-centric clusters.

As we have previously discussed, the ability to leverage the parallelism of a multi- or many-core architecture will play a key role in guaranteeing improved performance and interactive data analysis and visualization. Scout's runtime software layers attempt to achieve this goal by providing a low-level set of fundamental operations that can be combined to provide more advanced capabilities. These routines are written to exploit parallelism at both the node and cluster levels and are based upon the instructions introduced by Blelloch, and other parallel programming languages, for data parallel computing [15]. For example, consider the following set of steps required to prepare for the rendering of a traditional scatter plot comparing two variables and coloring the resulting points by a third field.

- The classic `min` and `max` reductions must be used to find the range of values within all three input variables. As was presented in figure 3, Scout adds support for a `range` operation that combines both the minimum and maximum value reductions into a single operation.
- In order to color the points by the third variable, each point must also maintain the knowledge of its location in the computational grid. This generated set of coordinate values closely mimics Blelloch's single-dimensional `index` operation (Scout extend this to support the generation of multidimensional vectors of data). As an implementation detail, these coordinates serve as a lookup table within the graphics hardware allowing us to leverage the GPU's high-bandwidth memory system to provide improved rendering rates.
- The two variables that serve as data points along the $x$ and $y$ axes must be combined to produce a set of points that can be sent to the graphics hardware for display. Once again using Blelloch's terminology, this operation is a *merge* and can be can be implemented using a modified version of the `flag-merge` instruction (we will refer to this operation as `merge`). (The use of Blelloch's terminology is helpful for discussion, but it does not necessarily provide the details of an efficient implementation on today's CPU or GPU hardware.)

We are actively working on developing efficient versions of these operations on emerging architectures. Scout's runtime layer supports execution on multicore systems ranging from a single desktop to clusters. Figure 4 shows the performance results for running on a single node with eight processor cores and 1 core on each of 16 nodes (top images). These results were run on a cluster containing dual quad-core 2.0 GHz Opteron processors and NVIDIA Quadro FX 5600 graphics cards with 1.5 GB of video memory. As expected, the simplicity of these operations allows them to scale well and we are able to achieve an interactive scatter plot functionality for moderately sized datasets (see the bottom right image of figure 4). This is built directly on the foundation that will eventually enable the scatter plots to be programmatically controlled in the same manner as the examples presented above and the rendering of point data previously presented [11]. The fundamental issue with achieving reasonable multicore performance is avoiding the impact of memory contention and various systems-level issues such as affinity and system noise. This is clearly shown by the difference in scaling behavior between the eight cores within a single node and the same effect across 16 nodes with multiple sets of active cores (bottom left image of figure 4).

In the current implementation the GPU's primary role is rendering, which is the current bottleneck in overall performance. The key challenge in shifting the GPU to a role of a computational coprocessor is to amortize out the costs of data movement between the CPU and GPU memories. Current benchmarks show that the bandwidth for these transfers are approximately 2.0 to 2.5 GB/sec depending upon the data sizes and recent additions to the CUDA API now allow for asynchronous communication [16]. We are actively working on overlapping transfers between both CPU and GPU computations to increase the level of concurrency during the operation of these operations – we believe this is one of the key issues to achieving a significant percentage of peak performance for multi- and many-core processors. Finally, we are actively working on incorporating the results of the CUDA Data Parallel Primitives Library that directly relate to GPUs and Blelloch's fundamental set of operations [17, 18]. This concept of a

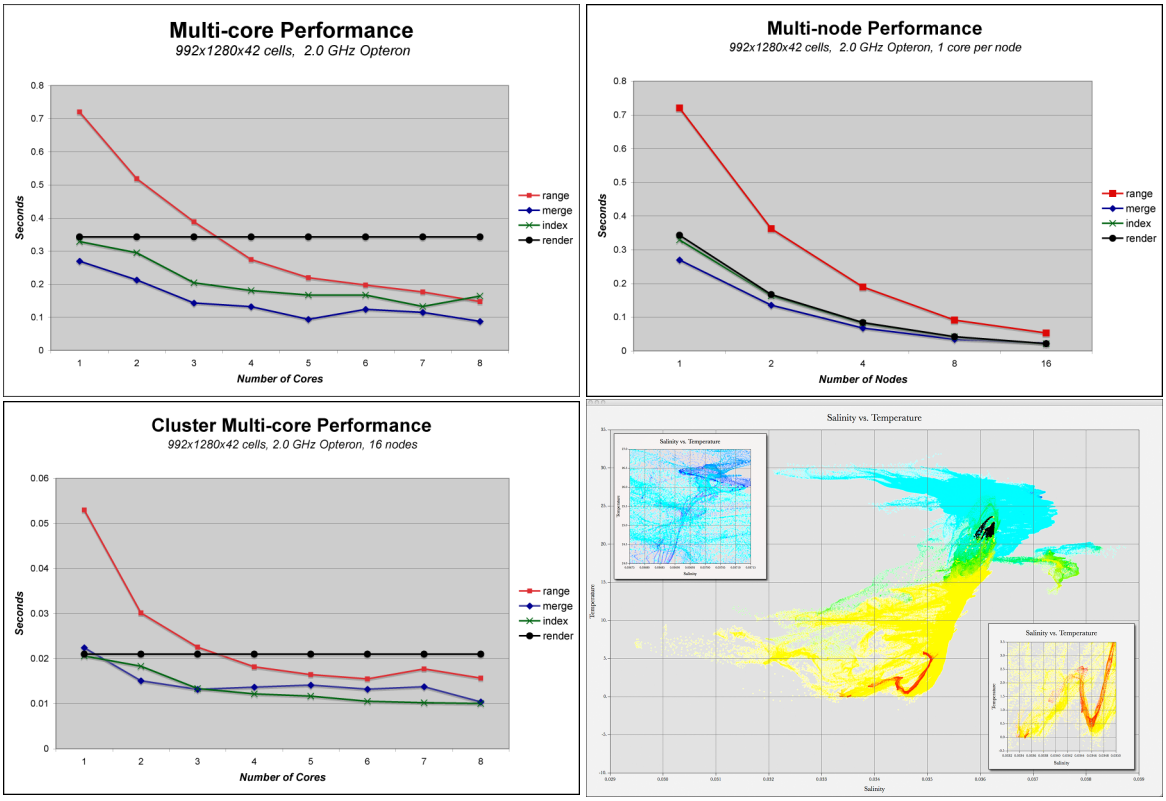


**Figure 4.** Performance scaling comparison for the computational portions of the scatter plot operations on a single node with an eight core CPU (top left), 16 cluster nodes using only a single core (top right), and 16 nodes with a varying number of cores in use. Although the benchmarked operations are extremely simple, the results show the clear disadvantage of memory contention for the multi-core, single-node, timings. In addition to the basic `range`, `merge`, and `index` operations the performance graphs include rendering time for comparison. The scatter plot tool built on these fundamental operations is shown in the lower right image. The scatter plot shows ocean temperature vs. salinity.

fundamental set of operators for programming complex, parallel systems will serves as a building block for further, higher-level operations related to the analysis of large-scale data.

### *4.2. Multivariate Example for Climate Modeling*

Scientific modeling of the world's oceans, specifically the North Atlantic and Arctic Oceans, is a topic of current research. Complex systems of differential equations govern the computations performed in the model, resulting in large, high-dimensional spatio-temporal data sets. These data contain information regarding the physical aspects of the ocean circulation (such as temperature, salinity, and current velocity), as well as variables describing abundances of nutrients (such as nitrates and silicates), dissolved gasses (such as oxygen and carbon dioxide), and both phytoplankton and zooplankton. This makes size and number of dimensions to consider confounding factors in both validating the operation of the model and analyzing the data it produces.

In order to verify the model behaves properly, scientists must manually inspect each dimension of the data. This examination must take into account not only the underlying mathematical behavior, but the individual data being used as input to the computation. Even for small and relatively simple models, this process is daunting and time-consuming. Existing tools do little to mitigate the problems associated with scientific model verification. With the size and complexity of simulations increasing as more powerful computing equipment is made available, a new suite of tools and verification methods is required.

While powerful analysis tools can be leveraged to form meaningful hypotheses from a simulation, equally powerful tools must be developed to both analyze and ensure the data produced by the code accurately portrays the underlying mathematical model. Tools commonly used to visualize and otherwise process simulation results are often ineffective when verifying the accuracy of a model. However, an environment equipped with a programmable interface, such as Scout, provides a flexible platform on which both visualization and verification can operate with a minimal amount of programming by the scientist.

One method to verify the proper behavior of a simulation is to examine the correlations between variables as the simulation progresses in time. These correlations can then be related back to the original mathematical description of the simulation in order to verify relationships between the variables in the model. In order to quickly find correlations between the many dimensions of such a large dataset, a fast and robust method must be used. While image-based correlations can be computed using each dimension of the data, this process is computationally expensive [19]. However, as the goal of this work is to reduce the number of dimensions of interest to a manageable subset rather than to perform pattern matching and recognition, a faster approximation can be formulated.

The examination and manipulation of an image's spectral properties is common in the field of image processing [20] . Since the frequency content of an image or time series is both unique and easy to compute using various Time-Frequency Representations (TFRs),it is possible to compute correlations between dimensions of a datum taking only their representations in the frequency (or time-frequency) domain into consideration. This approach has several advantages over the more straightforward image-space correlations common to pattern matching and recognition. Firstly, TFRs are agnostic to the range of the input data - no normalization of the input or removal of any data masks is required to adequately correlate any two dimensions. Secondly, the domain over which correlations must be computed is smaller than the image domain, offering an increase in processing speed. Finally, correlation computations are composed of several atomic, parallel operations. This natural parallelization is well-used by the Scout data-parallel programming language, making it accessible to the scientists analyzing the data.

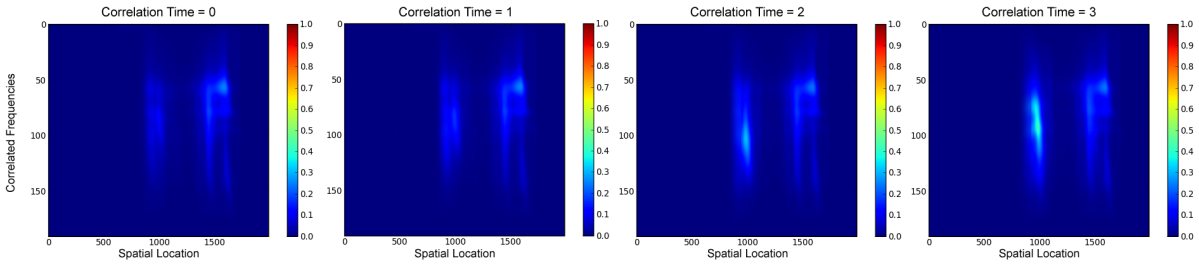


**Figure 5.** Correlation study performed between the presence of Diatom-based Carbon and $SiO_3$ dissolved in the water. As can be seen, the two data are correlated throughout time. By holding one dimension's time step constant, a relationship in time can be seen as the degree of correlation changes as time steps are processed. Artifacts due to land masses have been removed. The x-axis represents the spatial location of the correlation while the y-axis describes the correlated frequencies.

Figure 5 examines a small subset of correlation results that describe the relationships between two variables in a North Atlantic Oceanographic simulation. It should be noted here that, while the computation of image-space correlates still takes place, the smaller domain under consideration substantially accelerates the computation when compared to attempting to correlate the original input volumes. Additionally, the data-parallel nature of Scout further enhances the speed in which the results are computed while the direct programming of the system provides scientists a means to ensure all calculations performed are meaningful. This figure depicts a correlation over time between two variables in the input data set across a single spatial dimension. As the vertical dimension here is frequency, it is

inferred that the correlation is not unique to a single frequency, but rather a frequency band indicating that the spatial data is not changing at a uniform rate. While, in this case, only the horizontal axis of the resulting image represents a spatial dimension, Fourier-based transforms are often easily applied to any number of dimensions, allowing correlations to be found between any set of spatial or temporal dimensions. However, using image-based correlates, we are able to describe the *lag* of the correlations. This feature is useful in correlating variables changing together yet isolated from each other in space.

As in Section 4.1, the functional building-blocks of this example tie closely to the data-parallel operators outlined by Blelloch. With the exception of the time-frequency and correlation calculations, which are easily parallelized, this method can be expressed in terms data-parallel operations as follows:

- Extract the variable to which all others should be correlated against using Scout's range selection operator (see the example code in figure 2).
- For each vector (in parallel) in the selected dimension (time or space), compute a time-frequency representation of it.
- For each corresponding vector (again, in parallel) compute the time-frequency representation.
- Compute the correlation between each corresponding vector.
- Find the maximum correlation value and its location using `max` and `max-index` reduction operators.
- Sort the resulting correlations based on their value and select the $N$ most highly correlated variables to further explore.

These full set of operations, and any necessary support functionality (such as mean and standard deviation computations), are currently being added to Scout and will be exposed to the language for direct programmability by end users.

## 5. Conclusion

As the amount of information being generated and acquired is increasing in both size and complexity, new systems must be developed to handle the challenges presented by these growing datasets. Recently, CPU architectures are reaching physical design limits in terms of power consumption and heat dissipation. To solve this, manufacturers are producing CPUs with multiple cores, or processing units, on a single die. This new generation of architectures comes with increased performance but increased programmatic complexity. The Scout programming language and environment provide scientists with a variety of scalable, atomic, functions that can be used to scale their explorations to take advantage of the computational resources available to them, from a single desktop workstation, to a cluster consisting of hundreds of processors. In addition, this system combines the operations needed for both qualitative and quantitative visualization and analysis. Our goal is to enable scientists to more effectively explore their data in its entirety, without the need for sub-sampling or cropping large portions of their data. We hope this will eventually improve the overall scientific workflow by improving the process of gathering observable, empirical and measurable data to assist in the formulation and testing of hypotheses.

## References

[1] AMD 2008 Amd stream computing http://ati.amd.com/technology/streamcomputing/index.html
[2] NVIDIA 2008 Nvidia tesla c1060 computing processor http://www.nvidia.com/object/tesla_c1060.html
[3] Owens J D, Houston M, Luebke D, Green S, Stone J E and Phillips J C 2008 *Proceedings of the IEEE* **96** 879–899
[4] Goddeke D, Wobker H, Strzodka R, Mohd-Yusof J, McCormick P and Turek S 2008 *International Journal of Computational Science and Engineering* to appear
[5] GPGPU 2008 General-purpose computation using graphics hardware http://gpgpu.org
[6] Kahle J A, Day M N, Hofstee H P, Johns C R, Maeurer T R and Shippy D 2005 *IBM Journal of Research and Development* **49** 589–604
[7] Python 2008 Python programming language http://www.python.org
[8] Python 2008 Matplotlib / pylab – matlab style python plotting http://matplotlib.sourceforge.net/

[9] McCormick P S, Inman J, Ahrens J P, Hansen C and Roth G 2004 *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA: IEEE Computer Society) pp 171–178 ISBN 0-7803-8788-0
[10] McCormick P S, Inman J T, Ahrens J P, Mohd-Yusof J, Roth G and Cummins S 2007 *Parallel Computing* **33** 648–662
[11] Ahrens J, Heitmann K, Habib S, Ankeny L, McCormick P, Inman J, Armstrong R and Ma K L 2006 *Journal of Physics Conference Series* **46** 526–534
[12] Thinking Machines Corporation 1991 *C* User's Guide*
[13] Thinking Machines Corporation 1991 *CM Fortran User's Guide*
[14] Wikipedia 2008 Hsl and hsv http://en.wikipedia.org/wiki/HSL_color_space
[15] Blelloch G E 1990 *Vector models for data-parallel computing* (Cambridge, MA, USA: MIT Press) ISBN 0-262-02313-X
[16] NVIDIA 2008 Learn more about cuda http://www.nvidia.com/object/cuda_learn.html
[17] Sengupta S, Harris M, Zhang Y and Owens J D 2007 *Graphics Hardware 2007* pp 97–106
[18] Harris M, Sengupta S and Owens J D 2007 *GPU Gems 3* ed Nguyen H (Addison Wesley) chap 39, pp 851–876
[19] Castleman K R 1996 *Digital image processing* (Upper Saddle River, NJ, USA: Prentice Hall Press) ISBN 0-13-211467-4
[20] Gonzales R and Woods R 1992 *Digital Image Processing* (Addison-Wesley)