High Performance, Three-Dimensional Bilateral Filtering

E. Wes Bethel[1,2]

[1]High Performance Computing Research Department,
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

[2]Institute for Data Analysis and Visualization,
University of California, Davis,
1 Shields Ave, Davis, CA, USA, 95616.

5 June 2008

Contents

Abstract

Image smoothing is a fundamental operation in computer vision and image processing. This work has two main thrusts: (1) implementation of a bilateral filter suitable for use in smoothing, or denoising, 3D volumetric data; (2) implementation of the 3D bilateral filter in three different parallelization models, along with parallel performance studies on two modern HPC architectures. Our bilateral filter formulation is based upon the work of Tomasi [11], but extended to 3D for use on volumetric data. Our three parallel implementations use POSIX threads, the Message Passing Interface (MPI), and Unified Parallel C (UPC), a Partitioned Global Address Space (PGAS) language. Our parallel performance studies, which were conducted on a Cray XT4 supercomputer and a quad-socket, quad-core Opteron workstation, show our algorithm to have near-perfect scalability up to 120 processors. Parallel algorithms, such as the one we present here, will have an increasingly important role for use in production visual analysis systems as the underlying computational platforms transition from single- to multi-core architectures in the future.

1 Introduction and Previous Work

Image smoothing, or denoising, is a fundamental operation in computer vision and image processing. One of the simplest approaches to smoothing is to perform averaging of nearby points to compute an estimate of the denoised signal. A "box filter" computes an estimate using equal weights for all the nearby sample points. A better estimate of the average would be to afford greater weights to nearby points and smaller weights to more distant points. The Gaussian low-pass filter performs such an averaging using a set of weights defined over a normal distribution such that points nearby the target sample point have a greater contribution to the average than points far away from the sample point. This type of smoothing is isotropic in the sense that the filter application is performed independent of the underlying signal. The result is that it smooths equally in all directions, which has the unfortunate side effect of blurring edges.

In contrast, anisotropic smoothing methods would, ideally, remove noise while preserving important features like edges. Perona [8] developed an anisotropic smoothing technique based upon diffusion. Diffusion-based smoothing methods, which are based upon the solution of partial differential equations, aim to detect region boundaries using a computationally expensive iterative method. The idea is to perform smoothing within, but not across, regions.

Bilateral filtering, as defined by Tomasi [11], aims to perform anisotropic image smoothing using a low-cost, non-iterative formulation. The idea is to smooth images by computing the influence of nearby points in a way that removes noise "within regions," and that does not have the undesirable property of smoothing edge features. This formulation uses a straightforward, tunable estimate for region boundaries: a Gaussian-weighted difference in signal, or photometric space. The idea is that where a sharp edge exists, there will be a large difference in signal. That estimate is combined with a traditional Gaussian-weighted distance function to lessen the contribution from pixels distant in both geometric and signal space.

In bilateral filtering, the output at each image pixel $d(i)$ is the weighted average of the influence of nearby image pixels $\bar{i}$ from the source image s at location i. The "influence" is computed as the product of a geometric spatial component $g(i, \bar{i})$ and signal difference $c(i, \bar{i})$.

$$d(i) = \frac{1}{k(i)} \sum g(i, \bar{i}) c(i, \bar{i}) \tag{1}$$

where $k(i)$ is a normalization factor that is the sum of all weights $g(i, \bar{i})$ and $c(i, \bar{i})$, computed as:

$$k(i) = \frac{1}{\sum g(i,\bar{i})c(i,\bar{i})} \tag{2}$$

While it is possible to precompute the portions of $k(i)$ contributed by $g(i,\bar{i})$, which depend only on the 3D Gaussian PDF, the set of contributions from $c(i,\bar{i})$ are not known *a priori* as they depend upon the actual set of photometric differences observed across the neighborhood of $c(i,\bar{i})$ and will vary depending upon the source image contents and target location i.

Tomasi defines g and c to be Gaussian functions that attenuate the influence of nearby points such that those nearby in geometric or signal space have greater influence, while those further away in geometric or signal space have less influence according to a Gaussian distribution. So,

$$g(i,\bar{i}) = e^{-\frac{1}{2}\left(\frac{d(i,\bar{i})}{\sigma_d}\right)^2} \tag{3}$$

Here, $d(i,\bar{i})$ is the distance between pixels i and $\bar{i}$. The photometric similarity influence weight $c(i,\bar{i})$ uses a similar formulation, but $d(i,\bar{i})$ is the absolute difference $\|s(i) - s(\bar{i})\|$ between the source pixel $s(i)$ and the nearby pixel $s(\bar{i})$.

The bilateral filtering approach – combining spatial and signal weights to provide a robust, anisotropic estimate of a smoothed signal – has proven flexible and adaptable to a broad set of applications. Jones adapts this formulation for use in mesh smoothing [7]. There, they replace the photometric difference component with one that measures the difference in facet normals in noisy meshes. More recently, the formulation has been extended for use in smoothing diffusion tensor magnetic resonance imaging data [6]. In that work, the authors replace the notion of photometric similarity with a metric suitable for measuring the dissimilarity of diffusion tensors. Their non-iterative technique combines weighted averages of diffusion tensors with a diffusion tensor dissimilarity metric. The authors also show the results of segmentation operations applied to unfiltered and filtered DTMRI data. In principle, their technique is applicable to 3D DTMRI data, though their results are for 2D data only.

In this work, we extend the original Tomasi formulation of bilateral filtering for use on 3D volumetric data. We use the same spatial and photometric estimates and weighting as in [11]. We apply this formulation to 3D medical data, and present results of that application. In addition, we present the results of implementing this formulation using three different parallelization models: POSIX threads [2] on a symmetric multiprocessor machine (SMP), the Message Passing Interface (MPI) [10] on both SMP and distributed memory machines, and Unified Parallel C (UPC) [4] on both SMP and distributed memory machines.

2 Implementation

2.1 Three Dimensional Bilateral Filtering

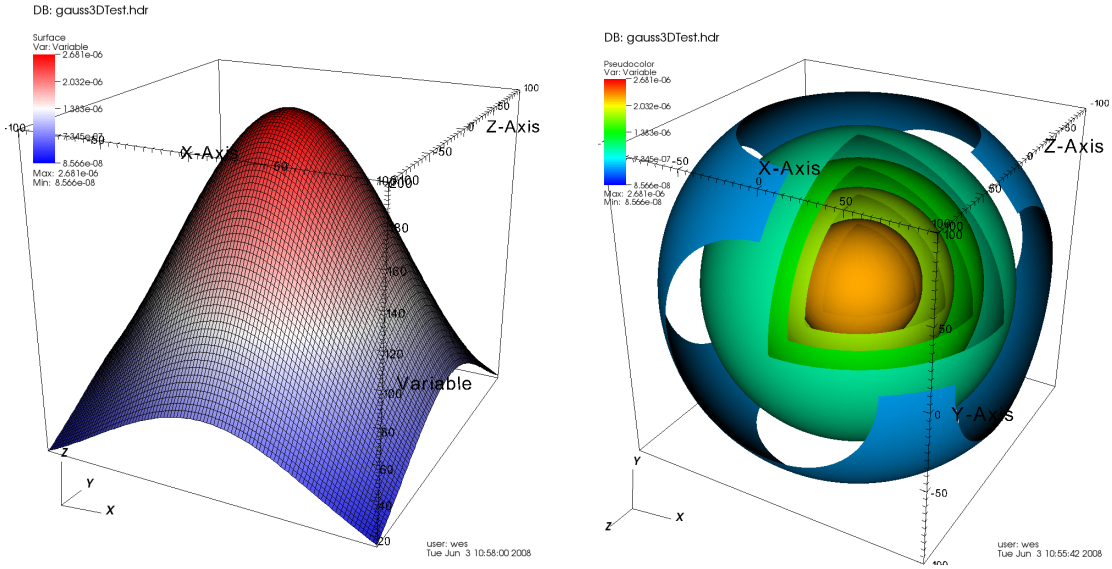
2.1.1 Gaussian PDFs

Our 3D bilateral filter is implemented using the same formulation as described by Tomasi [11]: the product of spatial and photometric difference components weighted by a Gaussian. Unlike Tomasi and subsequent work, we use a true 3D formulation and apply it to 3D volumetric data.

At the core of both the spatial and photometric filter components is a set of weights, based upon a Gaussian distribution, that give more emphasis to nearby points and less emphasis to distant points. The photometric weights are based upon a 1D Gaussian probability density function (PDF), whereas the spatial weights based upon a 3D Gaussian PDF. The 1D Gaussian PDF is computed as:

$$p = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(\frac{-(x-\bar{x})^2}{2\sigma^2}\right)} \tag{4}$$

For the photometric component, σ is a tunable parameter specified by the user. Larger values of σ give greater influence to pixels that are more distant in photometric space. For the spatial component, we use a constant value of σ that results in the sum of 3D Gaussian weights to be close to 1.0. The size, or number of samples in p, is fixed for the photometric component since we are processing input data in the range $[0 \ldots 255]$. The size of the 3D version of p, which is used to compute the spatial weight contribution of nearby pixels, is a tunable parameter specified by the user. Larger values result in more and more nearby points being included in the bilateral filter smoothing operations. 2- and 3-D versions of p are shown in Figure 1.



(a) (b)

Figure 1: These figures show visualization of the 3D Gaussian PDF (right) and a mid-volume orthogonal slice of the 3D Gaussian PDF (left). These images were used to verify the correct computation of the PDF during early stages of development.

2.1.2 Bilateral Filtering

The implementation of the bilateral filter for 3D volumetric data follows exactly from Equations (1), (2), and (3). For each output pixel $d(i)$, the contributions of pixels from the input image s are computed as the product of the 3D Gaussian weight of each nearby pixel as $g(i, \bar{i})$ and the 1D Gaussian weight of the photometric difference, $c(i, \bar{i})$, of the source and nearby pixel. Our work extends the original Tomasi formulation, which defined bilateral filtering on 2D images, for use on 3D volumetric data.

Our implementation consists of a three dimensional convolution kernel written in C. Here, the kernel, which is a 3D window, is positioned over the source 3D volume at each voxel location (i, j, k), and computes: (1) the product of the 3D Gaussian filter and the source voxel at the corresponding 3D kernel location; (2) the photometric difference between the target voxel at (i, j, k) and the source voxel at the corresponding kernel location, then (3) uses the results of (2) to perform a lookup into

a precomputed 1D table of Gaussian weights. The inverse of the sum of the products of (1) and (3) over the entire 3D kernel window is the normalization factor for this voxel location (Equation 2). The results are written into a 3D output image at the target location. Upon completion, the entire output image is written to disk in Analyze 7.5 format.

2.2 Parallel Implementations

With any parallel implementation, one of the fundamental design issues is how to partition the work across multiple processing elements (PEs). In this discussion, we use the term PE to refer to processing execution thread, regardless of how implemented: an execution thread on a multithread or multicore architecture, a separate pthread in POSIX threads, or an instance of a code running on each of several different distributed-memory processors. The usual approaches are task parallelism or domain parallelism. In domain parallelism, the problem is divided into smaller self-contained chunks and distributed across the PEs. Usually, this type of parallelism is implemented on P processors by having each processor work on $\frac{1}{P}$ of the data. In task parallelism, the problem is solved in "assembly line" fashion where the first PE solves the first part of the problem, then passes its results on to the next processor (or "station in the assembly line") for further processing. Most large-scale computations today use some form of domain parallelism to take advantage of parallel architectures, though there is interest in hybrid approaches, particularly on massively parallel machines composed of multicore processors. Such an approach has been used successfully in high performance, remote visualization, e.g., [1]. For this study, we implement domain parallelism using three different parallel programming models: POSIX threads, MPI and UPC.

2.2.1 Pthreads

Pthreads, which is an implementation of POSIX threads [2], provides an API for executing user code in separate execution threads that are scheduled by the operating system. The basic architecture of a threaded program is that it begins execution as a serial program, then explicitly creates new execution threads via the pthreads API. The main program will provide a function pointer to application code that will be executed in the separate execution thread. Usually, the main program and the threads will execute concurrently. At some point later, the main program can "join" the detached execution threads, then conclude as a serial program. The pthreads API provides mechanism for synchronizing threads in the form of semaphores and mutexes. In general, heap-based memory is visible to all execution threads, so pthreads is a convenient vehicle for implementing parallelism on shared memory machines like desktop platforms with multiple cores/processors.

In our implementation, we divide the problem domain evenly across the P execution threads: each of the P threads is responsible for computing $\frac{1}{P}$ of the final volume. The first thread is responsible for the first $\frac{1}{P}$ slices of output volume, the second thread is responsible for the next $\frac{1}{P}$ slices of output volume, and so forth. We read the data once in the main program prior to launching the P detached threads; all threads read from one array containing source data that is visible to all execution threads. Each execution thread writes its result into a different portion of the output volume array, which is also allocated off the heap by the main program and is visible to all execution threads. The main program acts as "work boss" by computing the portion of the problem domain for each thread, creating the threads, and then waiting for them to finish. After the P worker threads have completed, the main program writes the resulting smoothed image to an output file.

The pthreads implementation contains only a few dozen more lines of code than the serial implementation. The additional code is required to: launch and join the detached execution threads;

compute the domain decomposition and populate data structures containing the thread-specific work information; thread synchronization. This code is written in POSIX C, and will compile and run on any platform with a POSIX-compliant operating system, compiler, and implementation of pthreads.

2.2.2 The Message Passing Interface

The Message Passing Interface (MPI) is a parallel programming environment and runtime model [10]. It provides an API for performing various fundamental operations in parallel programs: point-to-point as well as collective communications and synchronization. Unlike pthreads program, where there is one main program that creates threads, an MPI program runs as main concurrently on P different processors. The duties of process launching and harvesting are the responsibility of the MPI runtime environment. Inside each MPI PE, it is the developer's responsibility to determine, via the MPI API, the number of PEs in play for this particular run as well as any PE's rank in the collective. Unlike pthreads, there is no notion of global shared memory: developers are responsible for moving data around amongst the MPI PEs (via the MPI API) as needed for a given application. Like POSIX threads, MPI is a specification, not an implementation. MPI has not been sanctioned by any standards body (unlike POSIX threads), but has become the *de facto* standard for parallel programming and execution on all modern HPC architectures. In contrast to POSIX threads applications, which will run only on shared-memory machines, MPI programs can run on either shared- or distributed-memory machines.

Our MPI implementation uses exactly the same domain decomposition strategy as our pthreads implementation. We divide the problem domain evenly across the P PEs: each of the P PEs is responsible for computing $\frac{1}{P}$ of the final image. The first PE is responsible for the first $\frac{1}{P}$ slices of output image, the second PE is responsible for the next $\frac{1}{P}$ slices of output image, and so forth. As a short-cut in this implementation, we have each PE load its own copy of the entire source volume into memory. This approach is viable in this case since the source data are relatively small – on the order of a few MB. A more robust implementation would read the data only once, and use the MPI API to copy portions to PEs as needed to fulfill their part of the work. After data is loaded, each PE performs the 3D bilateral filtering operation over its portion of the output domain. All PEs synchronize after concluding the filtering operation, then all PEs send their portion of the finished work via MPI_Send() calls to PE 0, which then writes the results into a file on disk.

Compared to the serial implementation, the MPI version of parallel bilateral filtering contains only a few dozen more lines of code. This extra code performs the following tasks: computes the domain decomposition to assign a separate portion of the problem to each PE, a small amount of inter-PE synchronization to obtain accurate timing information, and the data collection phase at the end of the parallel filtering operation. This code is written in POSIX C and uses MPI version 1.1, both of which are supported on virtually all modern HPC architectures.

2.2.3 Unified Parallel C

Unified Parallel C (UPC) is C with language extensions that implement global shared memory [4]. From the UPC website[1]: "The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD)

[1] http://upc.lbl.gov

model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor."

The primary advantage of Partitioned Global Address Space (PGAS) languages, like UPC, X10 [3], Titanium [13, 14], over parallel programming environments like MPI is that they alleviate the developer of explicitly moving data around between processors. Instead, these PGAS languages provide language-based semantical control over such memory management. A secondary advantage is that such memory management and movement can often be optimized "under the hood" to result in an application that performs significantly better than one where such management is manually coded. This expectation has proven true where the PGAS-based application was both faster and substantially smaller in terms of lines of code than the equivalent C++/MPI application [12].

Like MPI, a single instance of the code runs on all PEs. In other words, all PEs execute a main program. Also like MPI, the UPC runtime environment is responsible for managing launching and harvesting of all the PEs. In fact, a common implementation of the runtime environment for UPC is MPI: as part of the program compilation and linking process, the user code is translated into MPI code. However, the underlying memory management capabilities, which are not part of MPI, are provided by a layer under the application code. The particular implementation varies, but the most common is one called GASnet (Global Address Space networking)[2], which is a language independent, low-level networking layer providing network independent communication primitives tailored for implementing PGAS languages.

Our UPC implementation uses exactly the same domain decomposition strategy as our pthreads and MPI implementations. We divide the problem domain evenly across the P PEs: each of the P PEs is responsible for computing $\frac{1}{P}$ of the final image. The first PE is responsible for the first $\frac{1}{P}$ slices of output image, the second PE is responsible for the next $\frac{1}{P}$ slices of output image, and so forth. As with the MPI implementation, we take a "short cut" and have each PE read in its own copy of the data. After data is loaded, each PE performs the 3D bilateral filtering operation over its portion of the output domain. All PEs synchronize after concluding the filtering operation, then all PEs copy their portion of the finished work into an array that is visible to all PEs via the PGAS implementation. Then, PE 0 writes the results into a file on disk.

Compared to the serial version of the 3D bilateral filtering, the UPC version contains the least amount of "extra code" to realize a parallel implementation: a couple dozen lines of code are needed to perform the domain decomposition calculation, some inter-PE synchronization to obtain accurate timings, and the data gather stage. UPC is not a language standard; it is a research project that is a joint effort between the CS Department at UC Berkeley and the Future Technologies Group at LBNL. It (and GASnet) has been ported and tested on most modern HPC architectures.

3 Results

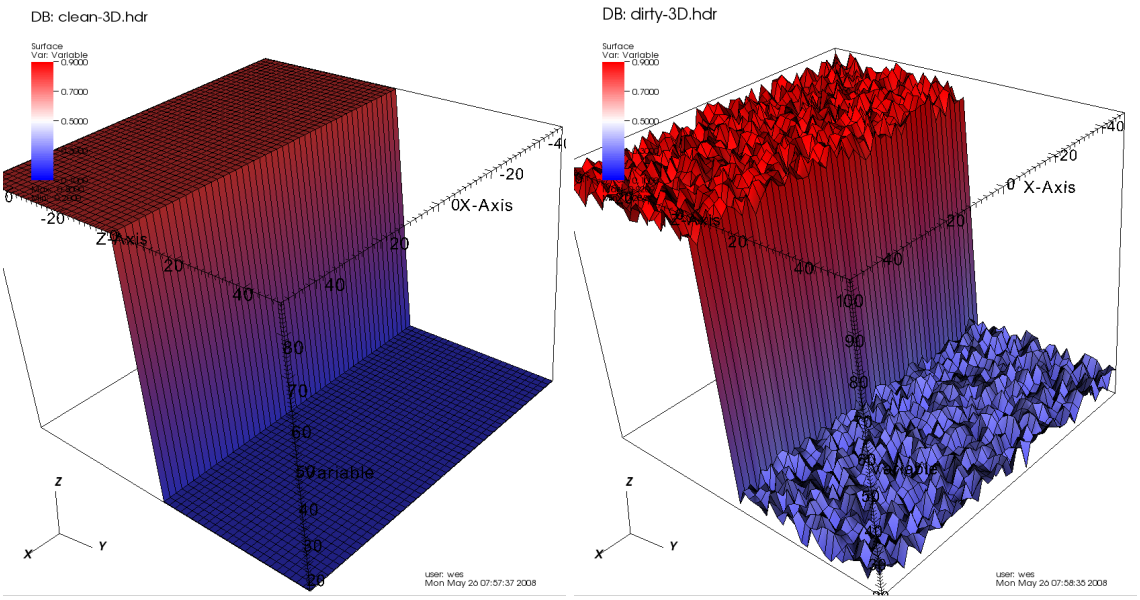
3.1 Three Dimensional Bilateral Filtering

3.1.1 Synthetic Data

We tested the 3D bilateral and Gaussian smoothing filters first on two synthetic datasets. Both are 3D volumes that are 50x50x50 voxels in size and consist of floating point scalars in the range 0.0 to 1.0. The volume is constructed such that the each of the high and low values occupy half of the volume; each half is spatially disjoint from the other. The first dataset, the "clean" dataset, consists of two values (0.20 and 0.80). We created a second, "dirty" dataset by adding Gaussian noise to the "clean" data.

[2]See http://gasnet.cs.berkeley.edu.

Figure 2 shows two images that illustrate the clean and dirty synthetic datasets. In these images, we extract a 2D slice from the clean and dirty synthetic datasets, then map scalar values on each slice to a "height" field. The left image shows a slice taken from the clean dataset. The right image shows a slice taken from the dirty dataset. These images show the sharp boundary between high and low values in the synthetic dataset.



(a) (b)

Figure 2: The image on the left is a 2D orthogonal slice taken from the "clean" 3D synthetic dataset then mapped to a height field. The image on the right shows a similar slice taken from the "dirty" synthetic dataset.

We applied 3D Gaussian smoothing to the synthetic dataset using a varying set of spatial filter widths. The results, shown in Figure 3, confirm the expectation that as we increase the size of the Gaussian filter, that the sharp edge in the synthetic dataset becomes blurred. This result is typical of isotropic filtering.

3.1.2 3D Medical Data

Next, we apply 3D Gaussian and bilateral filtering to a 3D medical dataset. Here, we run the smoothing operation for each filter over a range of parameter values. We vary the filter radius over the set of sizes [1, 2, 4, 8, and 16]. For the 3D bilateral filtering, we vary the width of the photometric filter over a set of parametric values that result in [5%, 10%, 15%, 20% and 25%] of the photometric difference falling within one standard deviation of the 1D Gaussian PDF. Since the input data is in the range $[0 \ldots 255]$, these target levels correspond to $\sigma = [13, 25, 38, 50, 63]$ since we need only compute and use half of the 1D Gaussian PDF.

Figure 4 shows a side-by-side comparison of raw data, 3D Gaussian smoothing, and 3D bilateral smoothing. That figure shows 2D slices from each data source, along with an x/y plot showing pixel intensity sampled at a particular scanline from each slice of data. In the raw data, we see sharp peaks and valleys, particularly at the corpus callosum. After 3D Gaussian filtering, the only

remaining features are the peaks corresponding to the *corpus callosum*[3], while all other features have been completely smoothed away. With the 3D bilateral filtering, we see virtually all of the distinguishing features of the raw data remain after filtering but with small-scale noise removed.

3.2 Performance of Parallel Implementations

Given our parallel implementation, in which we divide the workload evenly amongst the PEs and the parallel computation involves virtually no inter-PE communication, we expect relatively constant speedup as use more and more processors for a fixed problem size. In this strong scaling study, we compare the relative speedup of our parallel implementations on two different platforms.

The objectives for this performance study are as follows. First, we are interested in confirming the expected scalability characteristics of this algorithm on parallel machines. We expect near-perfect speedup as we use more and more processors. Second, we are using this opportunity to explore the relative ease and complexity of implementing a straightforward algorithm in three different parallel programming environments: POSIX threads, MPI, and UPC. Third, we are interested in examining the performance of two very new architectures: the Cray XT4 and a quad-socket, quad-core Opteron workstation. Fourth, we are interested in characterizing the runtime performance of this algorithm under varying input parameters.

For the first battery of tests, we compare the performance of our pthreads, UPC and MPI implementations on a workstation consisting of four quad-core 2.0Ghz AMD Opteron 8350 processors (16 total cores), sixteen total cores, 64 GB RAM, running SuSE 10.1, using gcc 4.2.1 at -O3 and UPC version 2.6.0. We run at varying levels of parallelism from 1 to 32 PEs.

For the second battery of tests, we compare the performance of our UPC and MPI implementations on a Cray XT4 system, which consists of 9660 dual-core 2.6 Ghz AMD Opteron 285 processors (19320 total cores), 4GB per node, PGI cc 7.0.7 at -O3 and UPC version 2.6.0. On the service nodes, this platform runs SuSE Linux. On the internal compute nodes, it runs a light-weight OS based on Linux, Compute Node Linux (CNL), which reduces system overhead, and is critical for the system to scale to very large levels of concurrency. We run at varying levels of parallelism from 1 to 120 PEs. Due to this machine's architecture and runtime environment, we cannot run pthreads applications at a level of concurrency exceeding the number of cores per node, which in this case is two cores. Therefore, there is no value in running the pthreads scalability study on this platform. The complete battery of tests consumed about 1900 CPU hours on the Cray XT4.

The test battery consists of running the 3D bilateral filtering algorithm on a 3D dataset varying the number of PEs and the size of the convolution filter. The number of PEs depends on the test platform: for the Cray, we vary the number of PEs over the range [1, 2, 4, 8, 16, 32, 64, and 120][4]. For the quad-core Opteron platform, we vary the number of PEs over the range [1, 2, 4, 8, 16 and 32].

3.2.1 Absolute Runtime

To give a feel for the absolute runtime of this algorithm on the Cray, Table 1 shows the elapsed runtime in seconds of the 3D bilateral filtering algorithm when executed on a 3D medical dataset that is 256x256x120 voxels in size. There, we report only the time elapsed during filtering; we do not include I/O time in the performance measurement. I/O time is negligible (fractions of a

[3]The corpus callosum is visible as the "C-shaped" structure in the middle of the cerebral cortex. It is an obvious white-matter structure that should not be smeared by smoothing operations.

[4]The source data size is 256x256x120, and we are dividing the problem domain along z-axis slices. Therefore, the maximum PE pool size that makes sense for this problem and using the slice-/slab-based decomposition is 120 PEs.

Filter Width	Number of PEs							
	1	2	4	8	16	32	64	120
r=1	3.43	2.28	0.89	0.44	0.23	0.12	0.06	0.03
r=2	13.03	6.52	3.29	1.65	0.88	0.44	0.22	0.11
r=4	74.13	37.20	18.88	9.47	5.04	2.49	1.22	0.58
r=8	461.08	232.00	120.51	60.62	32.15	16.07	8.17	4.10
r=16	3694.53	1906.47	1057.74	536.73	276.56	137.74	71.99	36.042

Table 1: Absolute runtime in seconds of the 3D bilateral filtering when run on the Cray XT4 at varying levels of parallelism and filter sizes. The 3D bilateral filtering algorithm shows near-perfect speedup as we add more processors: at the filter radius = 16 level, the single-processor configuration requires about half an hour to execute, while the 120 processor version completes in about 30 seconds.

second) as the dataset size is relatively small. We vary the size of the filter over the range [1, 2, 4, 8, 16] and at varying levels of parallelism, [1, 2, 4, 8, 16, 32, 64, 120]. The 3D bilateral filtering algorithm shows near-perfect speedup as we add more processors: at the filter radius = 16 level, the single-processor configuration requires about half an hour to execute, while the 120 processor version completes in about 30 seconds.

3.2.2 Scalability

In strong scaling, we keep the overall size of the problem fixed while increasing the processor count. In contrast, weak scaling measures performance while increasing the problem size for a given number of processors [5]. For our scaling tests here, we are measuring strong scaling characteristics: we keep the problem size fixed while varying the number of processors.

For our strong scaling study, we run each of the POSIX threads, UPC, MPI implementations over varying numbers of processors and filter width sizes. The runtime we report consists only of elapsed time for filtering: we omit I/O time, which takes only a fraction of a second as the problem size is relatively small. To account for potential runtime variance on a single system, each processor count and filter width configuration was run multiple times, varying the relative photometric difference weights. Varying those weights does not affect runtime since the same amount of computation is performed regardless of the photometric difference weights. However, running the test battery in this fashion was a convenient way to generate data we present in Section 3.1 and the Appendix. The strong scaling study results for both the Cray XT4 and quad-core Opteron system are shown in Figure 6.

For the Cray tests, we have the option of running jobs in a configuration that uses one or both cores on a given node: each node consists of a dual-core Opteron processor. Interestingly, we see a marked difference in relative performance of the UPC implementation depending upon whether we run using one or two threads per node (see Figure 7). Communication with the UPC developers reveals this behavior is a known problem with the current (v2.6.0) UPC implementation for the Cray XT4 system. The MPI implementation shows near-ideal scaling up to the maximum number of PEs for this test (see Figure 8). In contrast, the scalability for the single- and dual-thread UPC tests falls off at 64 processors. This behavior is also due to a bug in the UPC implementation on the Cray XT4.

For the Opteron tests, we see nearly identical scaling performance for all three parallel implementations. At 16 PEs, we see a falloff in scalability, which is expected since there are only 16 cores on this machine. Interestingly, between 16 and 32 processors, the scalability of the MPI and POSIX

threads implementation increase slightly, whereas the scalability of the UPC implementation falls off slightly. The cause of this feature is unknown at this time. Since the quad-core Opteron processor is very new (and since AMD releases new versions of the processor relatively frequently as they fix bugs), the community is still conducting investigating its performance.

4 Discussion and Future Work

The work we present here lays the groundwork for two separate publications that are reasonably within reach. The first focuses on a 3D bilateral filter, along with extensions, for use on smoothing 3D volumetric data. Earlier work that builds on the basic idea of the bilateral filter has proven successful when applied to mesh smoothing [7] and smoothing diffusion tensor magnetic resonance imaging data [6]. As presented, this work is probably not sufficiently novel for publication without a new metric that replaces the photometric difference metric originally defined by Tomasi in [11]. Some initial ideas in this direction include: (1) a metric based upon level sets: the idea would be to determine how well a given voxel fits into the level set of surrounding voxels; (2) a metric that determines how well the isocontour passing through a given voxel is in agreement with isocontours passing through surrounding voxels. The level set idea could possibly be implemented using a Fast Marching method, which is known to have a polynomial time, "non-iterative" solution [9]. An additional element required for a publication on filtering is comparison with the Perona anisotropic smoothing technique [8]. The basis for comparison would be both filtering quality as well as performance (absolute runtime and scalability).

The second potential publication has more of a visualization focus: the results we present here rely on a "chi-by-eye" comparison. We show side-by-side images of filtering results produced with different filtering parameters. In general, the subject of mesh-mesh comparative visual analysis is an open research area in the field of visualization. Several science application areas are in need of robust techniques in this space: e.g., climate modeling, where there is a desire to compare the results of ensemble model runs. Similar stories exist in many other fields of computational science: fusion energy, accelerator modeling, etc. New techniques for quantitative visual analysis of the filtering results we present here could likely be applicable to these other science application areas.

Because the Cray XT4 and quad-core Opteron systems are very new, a great deal of effort is currently underway, especially at the national laboratories, to better understand the performance of these systems. DOE has invested well over $300M in Opteron-based Cray systems at Oak Ridge National Laboratory and Lawrence Berkeley National Laboratory. Presently, these systems all use dual-core Opteron processors, with plans to upgrade to quad-core Opteron systems in the near future. There is a substantial amount of interest in developing a better understanding of the performance of these systems for different algorithms. The work we present here, which is essentially a stencil-based, memory-intensive application, will provide valuable data to that community.

5 Conclusion

This work has two primary thrusts: (1) developing and applying a formulation of the bilateral filter for use on smoothing 3D volumetric data, and (2) conducting a parallel performance study of that algorithm implemented in three different parallel languages on a pair of modern HPC platforms.

The filtering work has proven to be effective at performing anisotropic smoothing: noise is removed while features are preserved. The relative success of the of the bilateral filter in achieving that objective stems from a combination of weights that take into account both space and signal. Voxels that are far from a given target voxel will have relatively little influence in the smoothing

operation, while those that are closer have greater influence. The bilateral filter uses two metrics for computing "distant" and "nearby." The first is spatial distance, the second is difference in signal space. It is this latter metric that gives bilateral filtering its ability to preserve features characterized by a high gradient in signal, like edges. Our examples suggest that the width of the signal weight requires some tuning based upon features of a given dataset. For example, the x-y plot of a scanline of raw data shown in Figure 4 shows a profile of peaks and valleys corresponding to major structures in the brain. A knowledgeable user would be able to estimate a good set of photometric difference weights so as to preserve major features while removing noise. Our example in Figure 4 uses a filter spatial radius of four and a photometric difference weight of $\sigma = 15\%$. This combination seems to do a reasonable job of preserving features while removing noise within feature regions.

Our parallel implementations have shown near-perfect scalability. This result is expected since our implementation is "embarrassingly parallel:" there is no interprocessor communication required during parallel filtering. Our testing revealed known scalability bugs in the UPC implementation on the Cray XT4 whereby scalability degrades at higher level of parallelism. As more and more clinical computational platforms for data processing transition to multicore architectures, parallel implementations of algorithms, such as the ones we present here, will become increasingly important.

6 Acknowledgment

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We would like to thank Paul Hargrove, of the LBNL Future Technologies group, for providing access to an experimental quad-core Opteron SMP. Sample medical data for these studies was provided by Prof. Owen Carmichael, Department of Neurology, University of California, Davis, and the UC Davis Alzheimer's Disease Research Center.

7 Appendix A – Initial Visualization Results

This Appendix presents some early visualization results. In contrast to the examples earlier in this report, which use a grayscale transfer function, these examples use a transfer function with a set of colors intended to provide more "visual data" to help better illustrate how the filtering operation affects signal and noise.

The first set of images (Figure 9) show a set of 2D slices extracted from 3D dataset, while the second set (Figure 10 shows three orthogonal slices. These examples are intended to illustrate how the tunable parameters – filter spatial size and photometric difference weight – influence the smoothing operation.

References

[1] Wes Bethel, Brian Tierney, Jason lee, Dan Gunter, and Stephen Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 2000. IEEE Computer Society.

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

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[4] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC – Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[5] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), July 1988.

[6] G. Hamarneh and J. Hradsky. Bilateral Filtering of Diffusion Tensor Magnetic Resonance Images. *IEEE Transactions on Image Processing*, 16(10):2463–2475, Oct. 2007.

[7] Thouis R. Jones, Frédo Durand, and Mathieu Desbrun. Non-iterative, Feature-preserving Mesh Smoothing. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 943–949, New York, NY, USA, 2003. ACM.

[8] P. Perona and J. Malik. Scale-Space and Edge Detection Using Anisotropic Diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, 1990.

[9] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 2007.

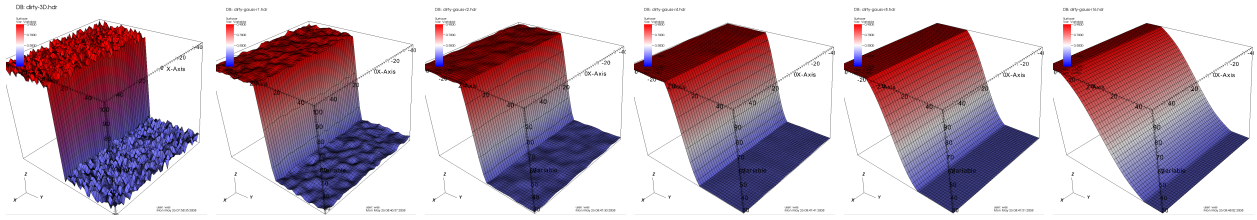
[10] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[11] C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 839, Washington, DC, USA, 1998. IEEE Computer Society.

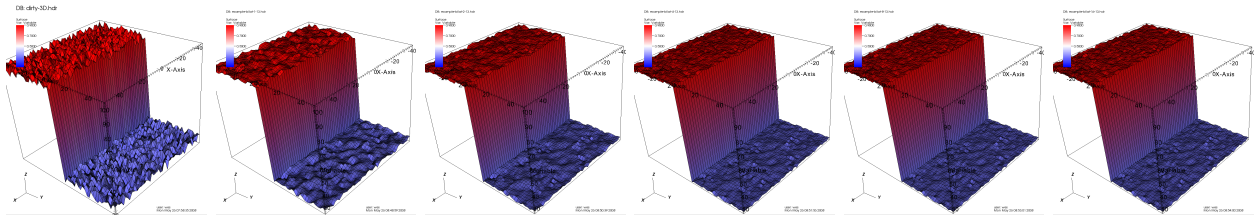
[12] Tong Wen and Phillip Colella. Adaptive Mesh Refinement in Titanium. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[13] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13), September–November 1998.

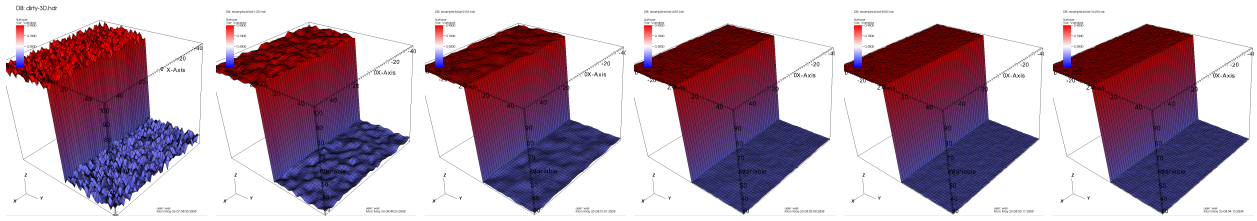
[14] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.



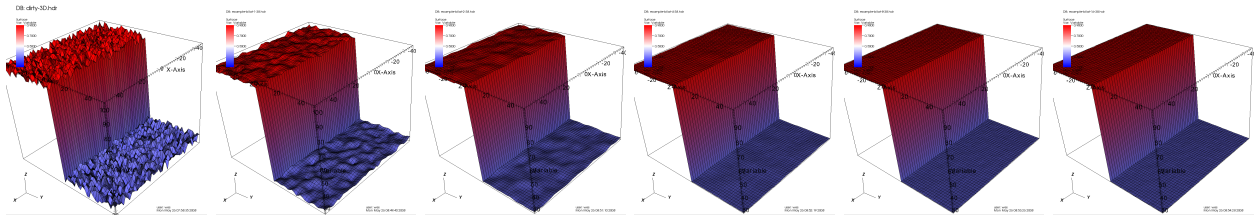
(a) 3D Gaussian smoothing. Left to right: original data, filter radius = 1, 2, 4, 8, 16.



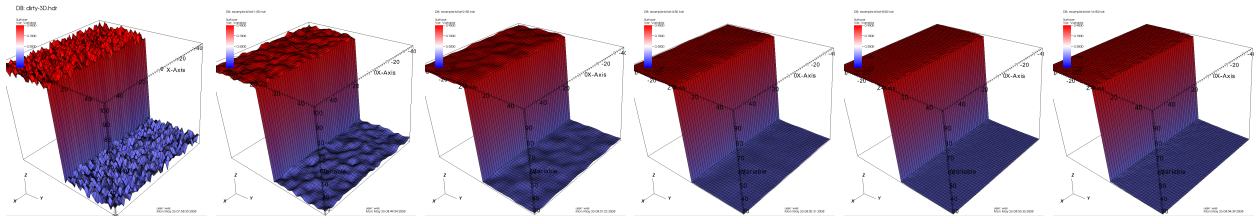
(b) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 5\%$.



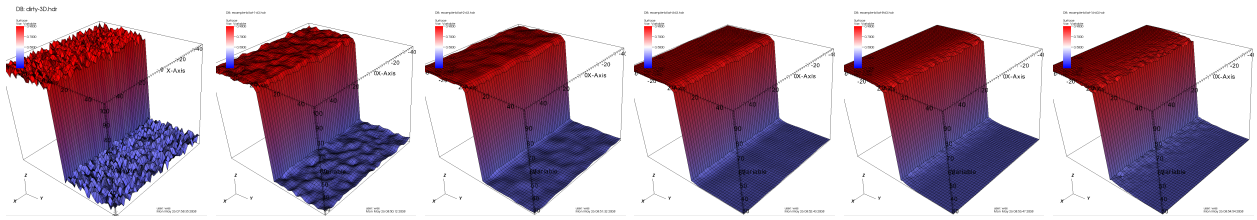
(c) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 10\%$.



(d) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 15\%$.

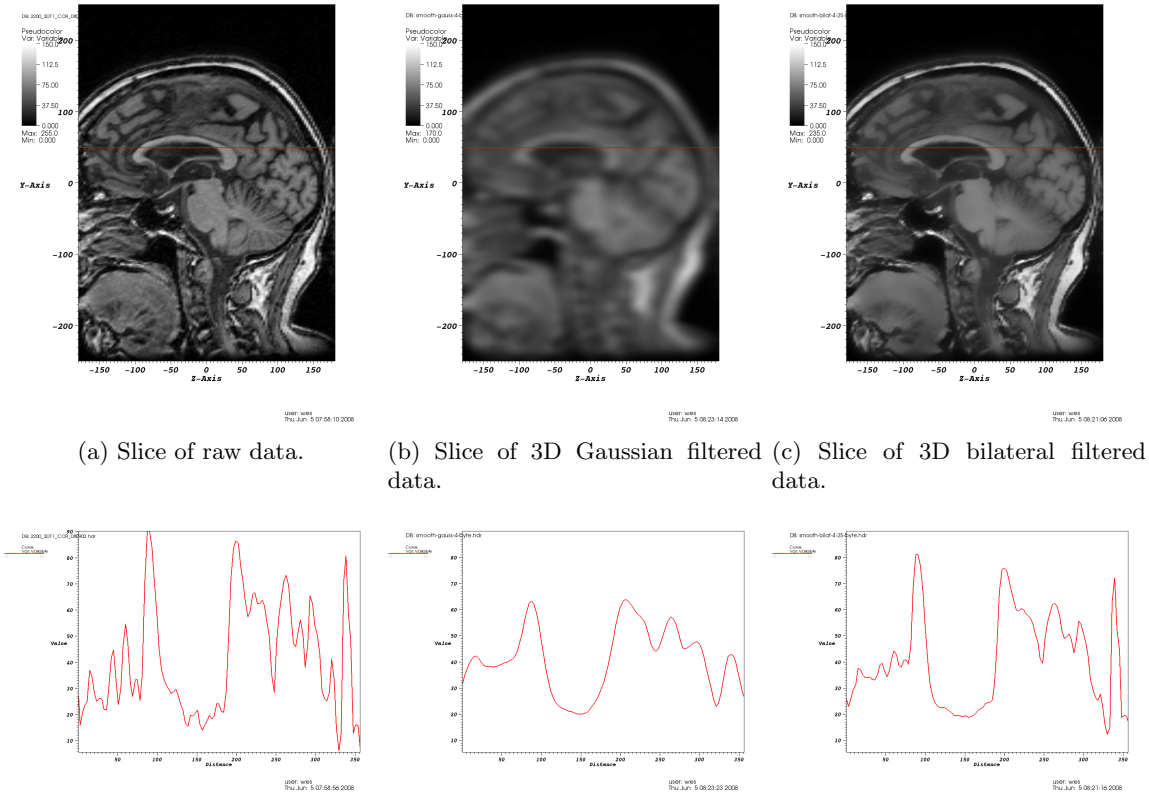


(e) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 20\%$.



(f) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 25\%$.

Figure 3: These images show the results of applying 3D Gaussian and bilateral smoothing to the noisy synthetic dataset. The top row shows results of 3D Gaussian smoothing at varying filter widths. The remaining rows show results of 3D bilateral filtering at varying filter widths and varying degree of photometric difference weighting.



(a) Slice of raw data.

(b) Slice of 3D Gaussian filtered data.

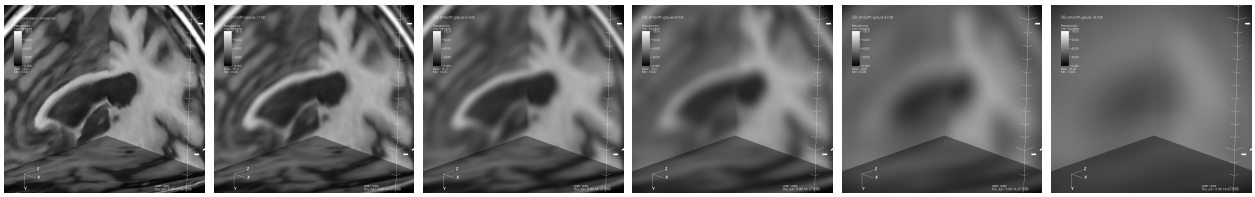
(c) Slice of 3D bilateral filtered data.

(d) Plot of raw data.

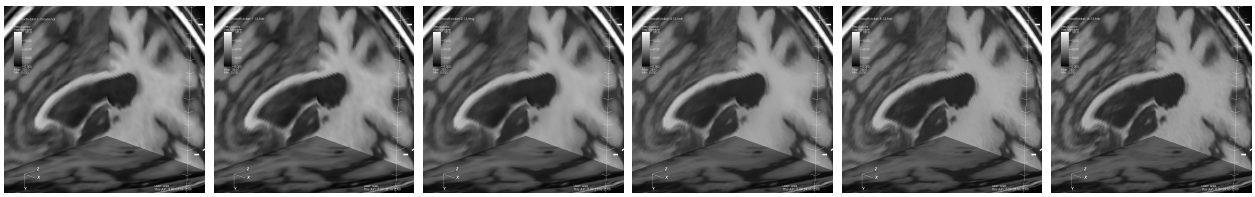
(e) Plot of 3D Gaussian filtered data.

(f) Plot of 3D bilateral filtered data.

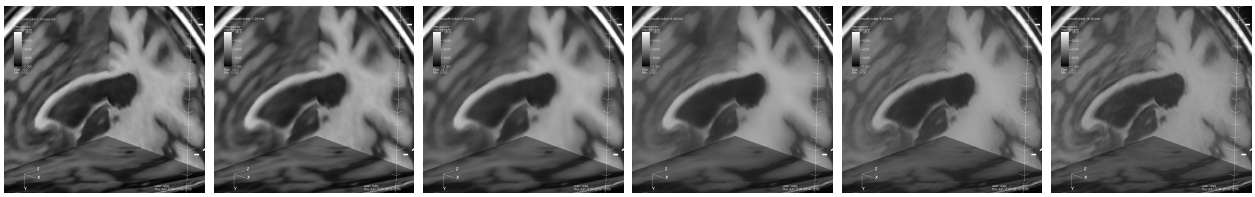
Figure 4: These images compare the results of 3D Gaussian and 3D bilateral smoothing. The top row contains slices of raw, Gaussian- and bilateral-smoothed data (left to right); the bottom row contains an xy plot of a row of pixels from each slice. We see the 3D Gaussian filter indiscriminately smooths data, whereas the 3D bilateral filter performs smoothing while preserving major features of the original dataset. For both 3D Gaussian and bilateral filtering examples above, the filter radius is equal to four. For the bilateral filtering's photometric difference is set to $\sigma = 15\%$.



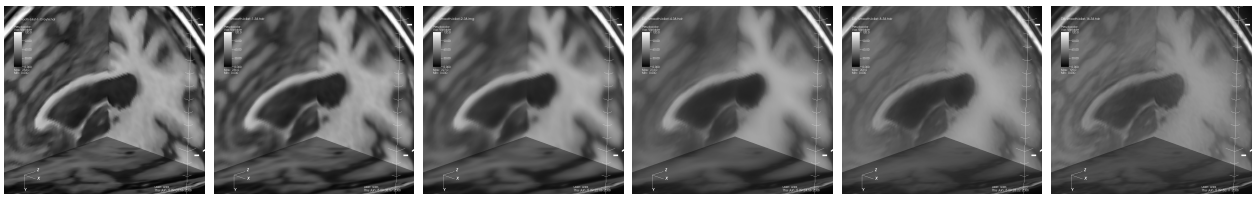
(a) 3D Gaussian smoothing. Left to right: original data, filter radius = 1, 2, 4, 8, 16.



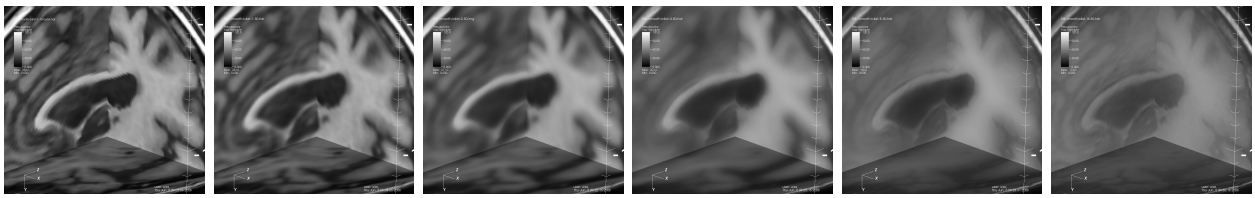
(b) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 5\%$.



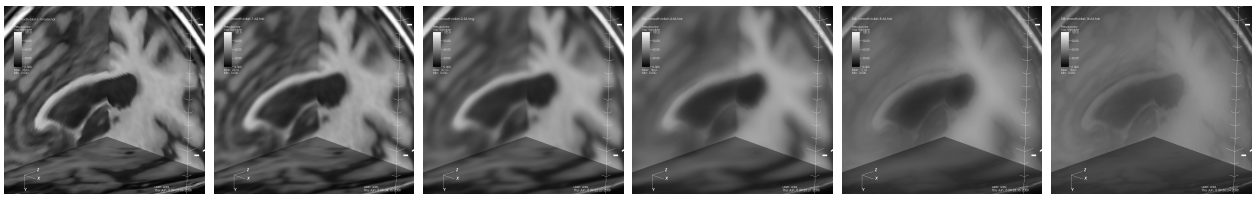
(c) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 10\%$.



(d) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 15\%$.



(e) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 20\%$.



(f) 3D Bilateral smoothing. Left to right: source data, filter radius = 1, 2, 4, 8, 16; photometric difference $\sigma = 25\%$.

Figure 5: These images show a zoomed-in view of three orthogonal slices of the 3D original and smoothed datasets. The top row shows results of 3D Gaussian smoothing at varying filter widths. The remaining rows show results of 3D bilateral filtering at varying filter widths and varying degree of photometric difference weighting.

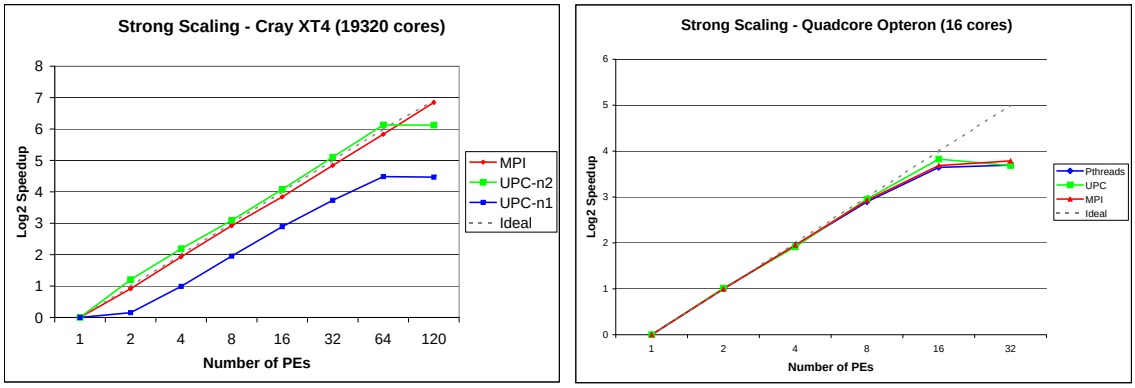


Figure 6: Strong scaling study results for the Cray XT4 (left) and quad-core Opteron systems (right). We are reporting scalability based upon the average runtime for all filter width levels at a given level of parallelism.

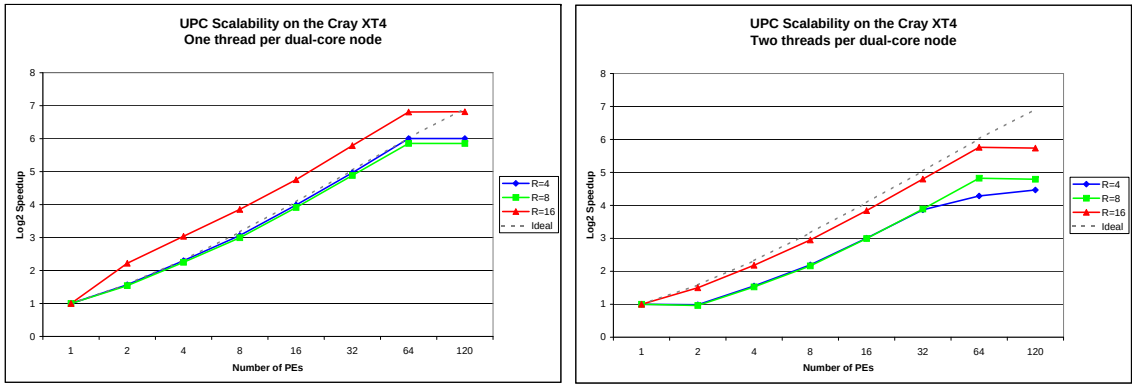


Figure 7: The UPC implementation on the Cray shows markedly different performance characteristics depending upon whether we run using one or two threads per node. When using one thread per node (left), we see near-perfect scalability up to 64 processors, and superlinear speedup at the $r = 16$ filter width. The superlinear speedup is mostly the result of caching effects: there is more opportunity for runtime benefit from due to caching effects with the larger filter width. Scalability falls off at 64 PEs due to a known bug in the Cray XT4 UPC implementation.

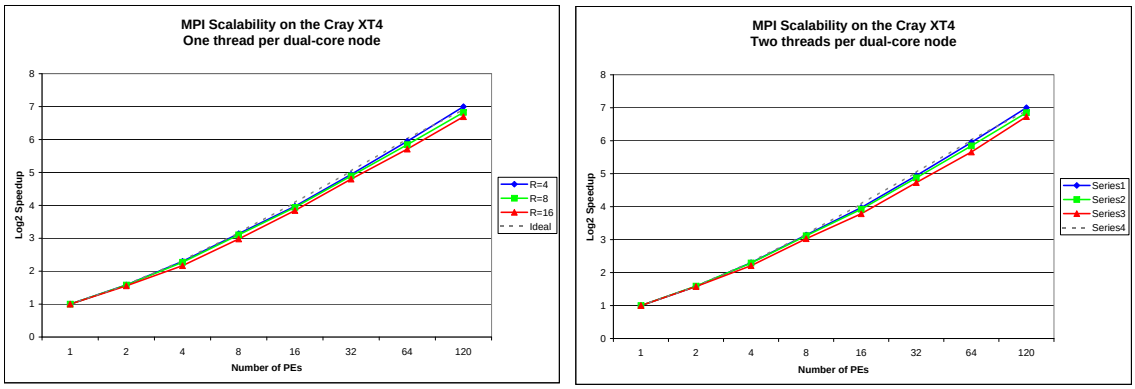
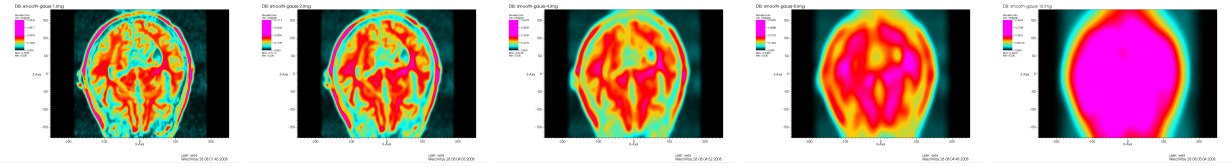
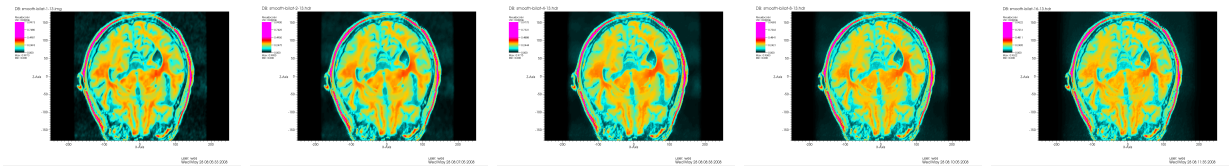


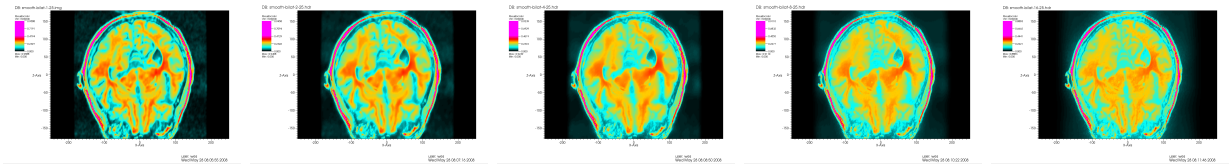
Figure 8: Our MPI implementation shows near-perfect scaling on the Cray XT4 regardless of whether we run with one thread per core (left) or two threads per core (right). Whereas the charts in Figure 6 report on average scalability over all filter widths, these charts show the scaling for three filter widths (4, 8, 16). The scaling characteristics for filter widths 1 and 2 are consistent with these results and are omitted for brevity.



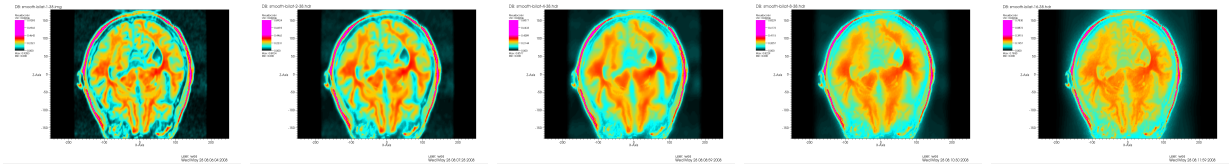
(a) Gaussian 3D filtering. Filter radius=1, 2, 4, 8, 16 (left to right).



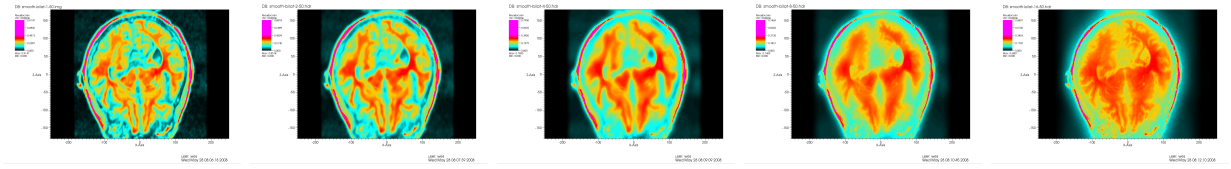
(b) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 5\%$.



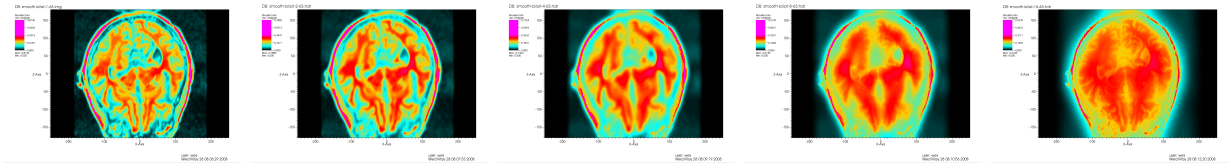
(c) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 10\%$.



(d) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 15\%$.

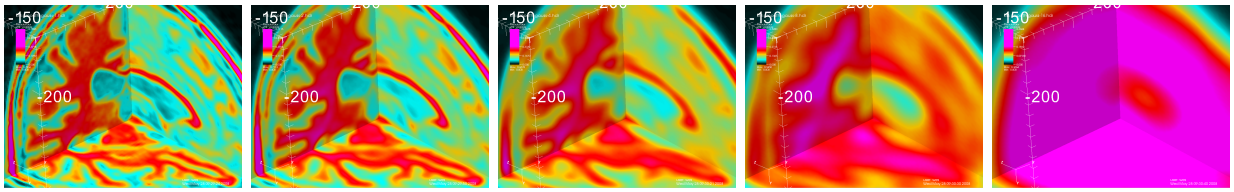


(e) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 20\%$.

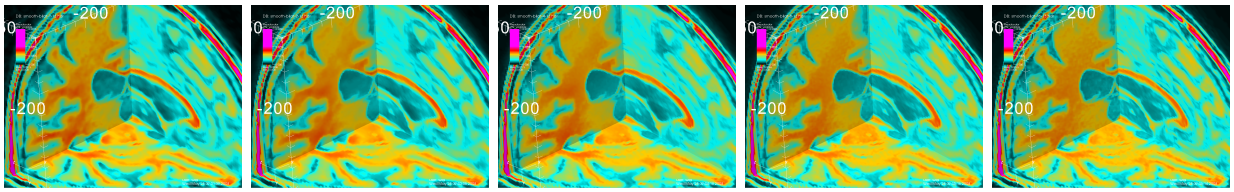


(f) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 25\%$.

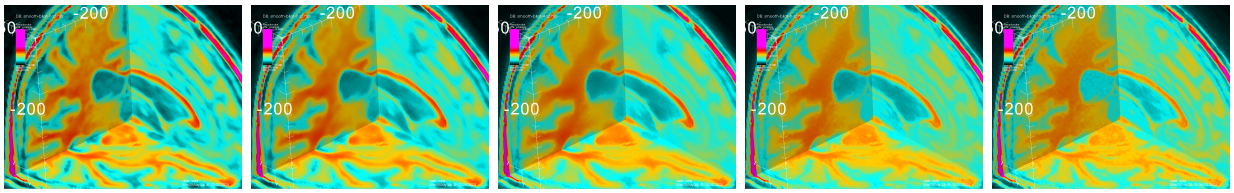
Figure 9: Comparison of 3D Gaussian and bilateral filtering. These images show a 2D slice from a smoothed 3D dataset. The colors in the transfer function where selected to induce a high amount of contrast, which helps to illustrate patterns of signal and noise across different filters and parameter settings.



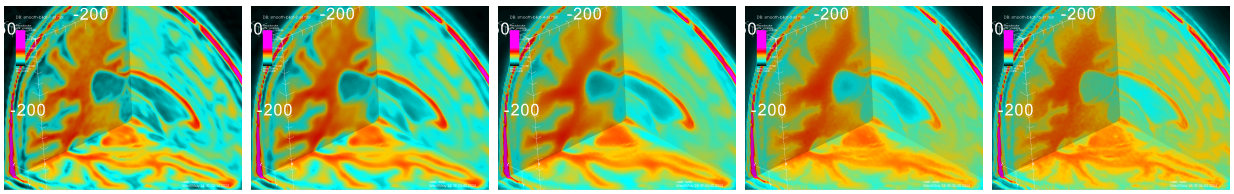
(a) Gaussian 3D filtering. Filter radius=1, 2, 4, 8, 16 (left to right).



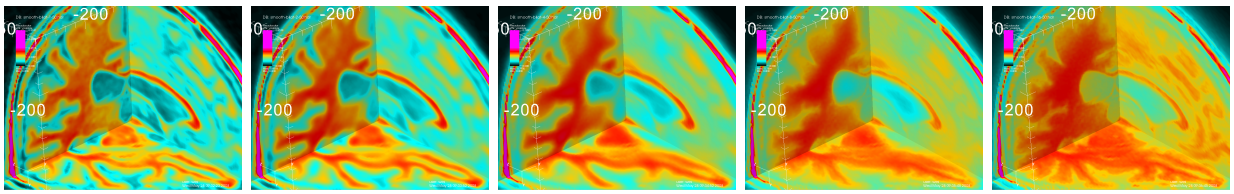
(b) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 5\%$.



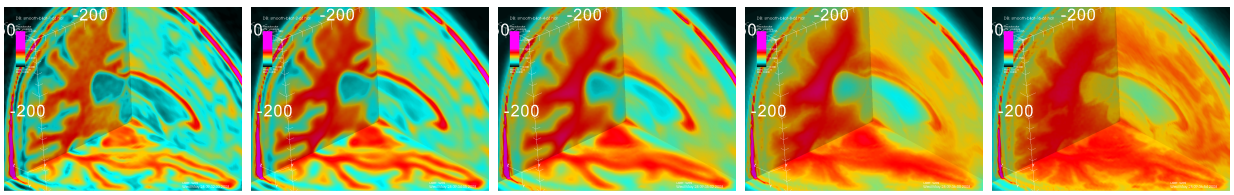
(c) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 10\%$.



(d) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 15\%$.



(e) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 20\%$.



(f) 3D Bilateral filtering. Filter radius=1, 2, 4, 8, 16 (left to right). Photometric difference weighting: $\sigma = 25\%$.

Figure 10: Comparison of 3D Gaussian and bilateral filtering. These images show a zoomed-in view of three orthogonal slices from a smoothed 3D dataset. The colors in the transfer function where selected to induce a high amount of contrast, which helps to illustrate patterns of signal and noise across different filters and parameter settings.