Parallel Colt: A High-Performance Java Library for Scientific Computing and Image Processing

PIOTR WENDYKIER and JAMES G. NAGY
Emory University

Major breakthroughs in chip and software design have been observed for the last nine years. In October 2001, IBM released the world's first multicore processor: POWER4. Six years later, in February 2007, NVIDIA made a public release of CUDA SDK, a set of development tools to write algorithms for execution on Graphic Processing Units (GPUs). Although software vendors have started working on parallelizing their products, the vast majority of existing code is still sequential and does not effectively utilize modern multicore CPUs and manycore GPUs.

This article describes Parallel Colt, a multithreaded Java library for scientific computing and image processing. In addition to describing the design and functionality of Parallel Colt, a comparison to MATLAB is presented. Two ImageJ plugins for iterative image deblurring and motion correction of PET brain images are described as typical applications of this library. Performance comparisons with MATLAB including GPU computations via AccelerEyes' Jacket toolbox are also given.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications; G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Deconvolution, FFT, inverse problems, iterative methods, multithreading, regularization, PET, motion correction

ACM Reference Format:
Wendykier, P. and Nagy, J. G. 2010. Parallel colt: A high-performance java library for scientific computing and image processing. ACM Trans. Math. Softw. 37, 3, Article 31 (September 2010), 22 pages. DOI = 10.1145/1824801.1824809. http://doi.acm.org/10.1145/1824801.1824809.

Research supported in part by the NSF under grant DMS-0811031.
Authors' addresses: P. Wendykier (corresponding author) and J. G. Nagy, Department of Mathematics and Computer Science, Emory University, 400 Dowman Drive, Suite W401, Atlanta, GA 30322; email: piotr.wendykier@gmail.com.

1. INTRODUCTION

Until relatively recently, improvements in CPU performance have been achieved by increasing the clock speed, execution optimization, and by increasing the size of on-chip cache. The *clock race* ended in 2003, when all chip manufacturers reached hard physical limits: increasing heat generation and power consumption, lack of suitable cooling hardware, current leakage problems, and increasing length of wire interconnects. October 2001 marks the beginning of a new era in CPU manufacturing when IBM released the POWER4 microprocessor, the world's first multicore processor. Since then, all new processors have been designed to consist of two or more independent cores on a single die. Six years later, in February 2007, NVIDIA publicly released CUDA SDK [NVIDIA Corporation 2009], a set of development tools to write algorithms for execution on Graphic Processing Units (GPUs). General-Purpose computation on GPUs (GPGPU) became available on virtually all desktop computers. Although software vendors have started parallelizing their products, the vast majority of existing code is still sequential. In practice this means, for example, that only one-fourth of a quad-core CPU (which is currently standard in a desktop computer) is utilized by a given program.

Parallel computing has been used for scientific computing applications since the 1960's, when the first supercomputers were developed. However, only recently have these programming paradigms become useful for software running on desktop and notebook computers. In this article we demonstrate the advantage of exploiting modern computer architectures in scientific computing with multithreaded programming in Java for applications in image processing. Our aim is to provide software that is efficient, flexible, and easy to use on affordable desktop and notebook computers.

Although Java was not designed to be a scientific computing language [Byous 2003], it has several unique features that are attractive for high-performance scientific computing. Because distributions are available for virtually all computing platforms, Java is an extremely portable programming language. In addition, starting in 2007 Java has become an open source project, allowing anyone to modify and adapt it to their needs. Java has native support for multithreading, and since version 5.0 [Sun Microsystems 2004] it is equipped with concurrency utilities in the java.util.concurrent package. Moreover, the performance of the latest version of Java (6.0) is comparable to the performance achieved by programs written in Fortran or C/C++ [Amerdo et al. 2008]. Finally, sophisticated imaging functionality is built into Java, allowing for efficient visualization and animation of computational results. This is especially important for our work in image processing, but is also useful in many areas of scientific computation, such as computational fluid dynamics.

However, it is important to recognize that because of certain design choices, there are also disadvantages to using Java in scientific computing. These include no primitive type for complex numbers, an inability to do operator overloading, and no support for IEEE extended precision floats. In addition, Java arrays were not designed for high-performance computing; a multidimensional array is an array of one-dimensional arrays, making it difficult to fully utilize

cache memory. Moreover, Java arrays are not resizable, and only 32-bit array indexing is possible. Finally, GPGPU is currently not possible in Java. There are libraries, such as JCuda [JCuda 2009], that provide Java bindings to CUDA, but they are only wrappers to underlying C code. To overcome these disadvantages, open source numerical libraries such as Colt [Hoschek 2004] or JScience [Dautelle 2007] have been developed.

To support our research collaborations in image processing, we implemented a fully multithreaded version of Colt, which we call Parallel Colt [Wendykier 2009d]. In this article we describe the basic features, functionality, and performance of Parallel Colt, which includes efficient handling of multidimensional arrays (e.g., for a 3-dimensional image), sparse matrix formats, dense and sparse matrix computational kernels, iterative solvers and preconditioners, trigonometric transforms, as well as unit testing and benchmarking tools. We also illustrate how Parallel Colt is used to solve large-scale inverse problems that arise in two imaging applications: iterative deconvolution and motion correction of brain images obtained from Positron Emission Tomography (PET).

Because MATLAB is widely used for scientific computing and imaging applications, we provide some comparisons between MATLAB and our Java implementations. MathWorks introduced multithreading in MATLAB R2007a, but even in the latest version (R2009b) the usage of multiple threads is limited. In particular, most of the linear algebra algorithms, such as matrix decompositions, are still sequential. This situation will probably change in the next release, due to the fact that the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [Buttari et al. 2007] is already available. GPU-based computations are available in MATLAB through a third-party toolbox called Jacket [AccelerEyes 2009]. Jacket does not introduce a new API, but instead allows programs written in the native M-Language to be automatically wrapped into a GPU-compatible form. Currently Jacket supports only NVIDIA graphic cards and, compared to standard MATLAB, its functionality is very limited. However, some of the algorithms can be compared with standard MATLAB, as well as with our Java implementations.

This article is organized as follows. In Section 2 we review some Java numerical libraries related to our work, and in Section 3 we present a technical analysis and the basic functionality of Parallel Colt. In Section 4 we describe two imaging applications where Parallel Colt is used: iterative image deblurring and motion correction of PET brain images. Concluding remarks and examples of other applications that use Parallel Colt are given in Section 5.

2. RELATED WORK

Although there are many open source Java packages for scientific computation, most existing libraries are targeted to solve problems from one particular field of study (as opposed to the comprehensive set of toolboxes in MATLAB). In this section we review four Java software projects, JScience, Matrix Toolkits for Java (MTJ), OR-Objects, and Universal Java Matrix Package (UJMP), which are most closely related to our work.

JScience [Dautelle 2007] is an open source package written by Jean-Marie Dautelle with the ultimate goal to "create synergy between all sciences (e.g., math, physics, sociology, biology, astronomy, economics, etc.) by integrating them into a single architecture." It supports multithreaded computations through the real-time programming library Javolution. Current features include modules for measures and units, geographic coordinates, mathematical structures (e.g., group, ring, vector space), linear algebra, symbolic computations, numbers of arbitrary precision, physical models (e.g., standard, relativistic, high-energy, etc.) and currency conversions. Nonetheless, JScience provides almost no support for image processing, and its linear algebra module is very limited, containing only the LU factorization. In addition, there is no class that represents a tensor (e.g., a 3D image array), no matrix subranging, and no Fast Fourier Transforms (FFT) [Cooley and Tukey 1965].

MTJ [Heimsund 2007] is a collection of matrices, linear solvers (direct and iterative), preconditioners, least squares methods, and matrix decompositions written by Bjørn-Ove Heimsund. This library is based on BLAS [Blackford et al. 2002] and LAPACK [Anderson et al. 1999] for dense and structured sparse computations and on Templates [Barrett et al. 1994] for unstructured sparse computations. By default JLAPACK [Doolin et al. 1999] is used, but MTJ can be configured to use native BLAS and LAPACK libraries (such as ATLAS [Whaley and Dongarra 1998]). Moreover, the library supports distributed computing via an MPI-like interface. However, MTJ does not supply multithreading, tensors, complex matrices, matrix subranging, and FFTs.

OR-Objects [DRA Systems 2000] is a collection of 500 Java classes developed by DRA Systems. It contains packages for linear programming, graph algorithms, matrix and linear algebra, numerical integration, probability and statistics, and geometry. Although OR-Objects is a freeware library, the source code is unavailable, which makes it much less attractive from our point of view. Moreover, analogous to JScience and MTJ, OR-Objects does not provide FFTs, tensors, complex matrices, and its multithreaded functionality is limited only to BLAS.

UJMP [Arndt et al. 2009] is a new project that aims to provide classes for storing and processing matrices with interfaces to external data sources (such as databases) and matrix libraries (including Parallel Colt). It allows to handle data that does not fit into main memory (the matrix size can be up to 2^{63} rows or columns). In addition, UJMP supports n-dimensional arrays and visualization methods for matrices. However, this package is also not designed for image processing. In particular, UJMP does not provide any parallel algorithms, complex matrices, and Fourier transforms.

3. PARALLEL COLT

In this section we describe Parallel Colt, a Java software that overcomes many of the deficiencies outlined in the previous section for JScience, MTJ, OR-Objects, and UJMP. There are two goals of Parallel Colt: (1) provide an open source Java library for high-performance scientific computing that utilizes modern hardware architectures; (2) provide an image processing engine with an emphasis on performance and usability.

3.1 Colt

Colt [Hoschek 2004] is an open source library for high-performance scientific computing in Java written by Wolfgang Hoschek at CERN. It provides efficient and usable data structures and algorithms for data analysis, linear algebra, multidimensional arrays, statistics, histogramming, Monte Carlo simulation, and concurrent programming. The project is currently inactive; the latest version (1.2.0) was released in September 2004. We have chosen to adapt Colt to fit our purpose of having a powerful computing engine for image processing. Our choice was motivated primarily by the fact that Colt has support for uniform, versatile, and efficient multidimensional arrays (matrices) [Hoschek 2000]. In particular, *views* operations defined on multidimensional arrays allow subranging, striding, transposition, slicing, index flipping, cell selection, as well as sorting, permuting, and partitioning of the elements. This is almost the same range of functionality as provided by MATLAB. In the rest of this section we summarize all the changes and new functionalities that we introduced in Parallel Colt.

3.2 Concurrency

Multithreading in Colt 1.2.0 is limited to a subset of BLAS routines: matrix-matrix and matrix-vector multiplications, as well as the generalized matrix scaling/transform. All other algorithms included in the library are sequential. Moreover, Colt uses Doug Lea's EDU.oswego.cs.dl.util.concurrent package for concurrency instead of the improved, more efficient and standardized classes (java.util.concurrent) which are included in a standard Java distribution since version 5.0. Concurrency in Colt requires setting a maximum number of threads before the first use, as opposed to Parallel Colt, where multithreading is enabled by default (if the number of available CPUs is greater than one). Java utility classes for concurrent programming contain the *cached thread pool* feature that we have found to be very useful. This type of pool creates new threads as needed, and reuses previously constructed threads when they become available, thereby improving the performance of programs that execute many short-lived asynchronous tasks. Because almost all element-by-element operations and BLAS routines can be split into asynchronous tasks, Parallel Colt uses the *cached thread pool* for low-level concurrency.

3.3 Multidimensional Arrays

In Colt, a single contiguous one-dimensional Java array is used to store elements of all dense 2D and 3D matrices. The elements of 2D matrices are addressed in row-major order and the elements of 3D matrices are addressed in (in decreasing order of significance): slice-major, row-major, and column-major order. However, there are two problems with this approach. First, matrix decomposition algorithms typically expect input matrices to use column-major order. Second, since Java array indices must be 32-bit integer values, the 1D array cannot contain more than 2^{31} elements, which is a significant limitation for large-scale problems. To overcome these difficulties, Parallel Colt provides additional data structures for dense matrices: 2D dense matrix addressed in

column-major order as well as 2D and 3D dense matrices where elements are stored in two-dimensional and three-dimensional Java arrays, respectively.

The original Colt project supports three types of 2D sparse matrices: row-compressed, tridiagonal, and the general sparse matrix that uses a hashmap to store the nonzero elements. However, it is beneficial for some applications to use alternative sparse storage schemes. Therefore, in Parallel Colt we have added a column-compressed sparse matrix (for fast column access) and a diagonal matrix (for more efficient computations involving only a single diagonal).

Another new and important type of object added to Parallel Colt is a multidimensional array of complex numbers. This object is essential for operations involving FFTs. Because there is no primitive type for complex numbers in Java, we decided to store an array of complex numbers as a one-dimensional array of doubles (or floats), interleaving the real and the imaginary parts. This type of storage guarantees much better performance than defining a new object that represents a complex number, and then storing an array of such objects.

There are many scientific computing problems where double precision is unnecessary; for example, in image processing a source image is often saved in a grayscale 8-bit format (integers from 0 to 255). From the computational point of view, single precision has two advantages over double precision: arithmetic operations are faster with single precision numbers and they require only half the storage of double precision numbers. All algorithms in Colt 1.2.0 that use floating-point numbers are implemented in double precision, in particular, only double precision multidimensional arrays are available. Therefore, in Parallel Colt we have added single precision equivalents to all double precision-based objects.

In addition to matrices holding floating-point elements, Parallel Colt fully supports matrices holding integer elements (both 32-bit and 64-bit versions). These type of objects are useful for processing RGB images, where the values of red, green, and blue channels are packed into single 32-bit integer values.

Colt is equipped with three different sorting algorithms: quicksort, mergesort, and binary search, which complement the java.util.Arrays class. Moreover, these algorithms are used to sort elements of multidimensional arrays. In Parallel Colt we have implemented a multithreaded version of quicksort that works both on arrays of primitive types and arrays of objects.

Finally, Parallel Colt's implementation of multidimensional arrays includes many additional methods, which are summarized in Table I.

3.4 Iterative Solvers

Once all types of sparse and dense matrices have been implemented, we have added to Parallel Colt a set of iterative solvers and preconditioners. The following solvers and preconditioners have been adapted from MTJ [Heimsund 2007].

Solvers [Barrett et al. 1994; Golub and Loan 1996]:
—BiConjugate Gradients (BiCG)
—BiConjugate Gradients stabilized (BiCGstab)
—Conjugate Gradients (CG)

Table I. Additional Methods in Parallel Colt

Matrix type	Method
All 1D	reshape
All 2D and 3D	vectorize
All real	getMaxLocation, getMinLocation, getNegativeValues, getPositiveValues, normalize
Dense 1D complex	fft, ifft
Dense 1D real	fft, ifft, getFft, getIfft, dht, idht, dct, idct, dst, idst
Dense 2D complex	fft2, ifft2, fftColumns, ifftColumns, fftRows, ifftRows
Dense 2D real	fft2, ifft2, fftColumns, ifftColumns, fftRows, ifftRows, getFft2, getIfft2, getFftColumns, getIfftColumns, getFftRows, getIfftRows, dht2, idht2, dhtColumns, idhtColumns, dhtRows, idhtRows, dct2, idct2, dctColumns, idctColumns, dctRows, idctRows, dst2, idst2, dstColumns, idstColumns, dstRows, idstRows
Dense 3D complex	fft3, ifft3, fft2Slices, ifft2Slices
Dense 3D real	fft3, ifft3, getFft3, getIfft3, getFft2Slices, getIfft2Slices, dht3, idht3, dht2Slices, idht2Slices, dct3, idct3, dct2Slices, idct2Slices, dst3, idst3, dst2Slices, idst2Slices

In the first column, "All" refers to all supported matrix data types, including single precision (complex and real), double precision (complex and real), 32-bit and 64-bit integers, etc.

—Conjugate Gradients squared (CGS)
—Generalized Minimal Residual using restart (GMRES)
—Iterative Refinement (Richardson's method)
—Quasi-Minimal Residual (QMR)
—Chebyshev iteration

Preconditioners [Golub and Loan 1996; Saad 1994; Vanek et al. 1996]:
—Diagonal (uses the inverse of the diagonal as preconditioner)
—Incomplete Cholesky without fill-in (ICC)
—Incomplete LU without fill-in (ILU)
—Incomplete LU with fill-in (ILUT)
—Symmetrical Successive Overrelaxation (SSOR)
—Algebraic Multigrid (AMG)

Besides the aforesaid solvers and preconditioners, Parallel Colt also supports the following preconditioned and nonpreconditioned solvers for ill-posed inverse problems:
—Hybrid Bidiagonalization Regularization (HyBR)
—Modified Residual Norm Steepest Descent (MRNSD)
—Conjugate Gradient for Least Squares (CGLS)

3.5 Linear Algebra

Multithreaded dense linear algebra in Parallel Colt is provided by JPlasma [Wendykier 2009b], which is our Java port of Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [Buttari et al. 2007]. An important matrix factorization for image processing applications is the Singular Value Decomposition (SVD), but currently PLASMA does not have support for it. Therefore, Parallel Colt implements two sequential SVD algorithms. One is the original Colt version, which is essentially a slightly modified Jama [Hicklin

et al. 2005] implementation, and the other is a divide-and-conquer routine from JLAPACK (`dgesdd`). Note that our present use of the SVD in image processing is within a Krylov subspace method that enforces regularization on a (small) projected linear system; see Chung et al. [2008].

Besides including JPlasma and JLAPACK in Parallel Colt, we have also added the following dense linear algebra operations: Kronecker product of 1D and 2D matrices (complex and real), Euclidean norm of 2D and 3D matrices computed as a norm of a vector obtained by stacking the columns of the matrix on top of one another, and backward and forward substitution algorithms for 2D real, upper, and lower triangular matrices.

Finally, we have implemented and included in Parallel Colt a Java version of the Concise Sparse Matrix Package (CSparse) [Davis 2006], which we call CSparseJ [Wendykier 2009a]. Although CSparseJ is not multithreaded, it provides a set of matrix factorizations (LU, Cholesky, and QR) that are much more efficient on sparse matrices than their dense equivalents. In the previous version of Parallel Colt, we used the same matrix factorization algorithms both for sparse and dense matrices (sparse matrices were converted to a dense form).

Currently Parallel Colt does not support linear algebra algorithms (except matrix-matrix and matrix-vector multiplications) for complex matrices.

3.6 Trigonometric Transforms

Trigonometric transforms, including the Discrete Fourier Transform (DFT) [Cooley and Tukey 1965], the Discrete Hartley Transform (DHT) [Hartley 1942], the Discrete Cosine Transform (DCT) [Ahmed et al. 1974] and the Discrete Sine Transform (DST) [Yip and Rao 1980], are important tools in image processing applications. To provide trigonometric transform functionality to Parallel Colt, we have integrated a library that we developed for this purpose, called JTransforms. We remark that all transforms are implemented as public methods in 1-, 2-, and 3-dimensional dense matrices (see Table I). In addition, they can be applied to matrix subranges.

JTransforms [Wendykier 2009c] is the first open-source multithreaded FFT library written in pure Java. The code was derived from the General-Purpose FFT Package by Ooura [2006] and from Java FFTPack [Zhang 2005]. Ooura's library is a multithreaded C and Fortran implementation of the split-radix FFT algorithm. In order to provide more portability both POSIX threads and Windows threads are used in the implementation. Moreover, the code is highly optimized and in some cases runs faster than FFTW [Frigo and Johnson 2005]. Even so, the package has several limitations arising from the split-radix algorithm. First, the size of the input data has to be a power-of-two integer. Second, the number of computational threads must also be a power-of-two. Finally, one-dimensional transforms can only use two or four threads. To overcome the power-of-two limitation we have adapted Zhang's Java code which is a straightforward translation of the mixed-radix algorithm from FFTPACK [Swarztrauber 2004]. Since Java FFTPack contains only sequential algorithms for 1D transforms (real and complex), we have implemented multithreaded 2D and 3D transforms. In the case of 1D transforms when the size of the vector is

not a power-of-two, JTransforms uses a sequential implementation. However this limitation does not affect the performance of multidimensional transforms because threads are used at higher levels. As a result, the current version of JTransforms can be used for arbitrarily sized data.

There are some important distinctions between our Java code and Ooura's C implementation. First, JTransforms uses a thread pool, while the original package does not. Although thread pooling in POSIX threads is possible, there is no code for this mechanism available in the standard library, and therefore many multithreaded applications written in C do not use thread pools. This has the added problem of causing overhead costs of creating and destroying threads every time they are used. Another difference between our code and Ooura's FFT is the use of *automatic multithreading*. In JTransforms (and in Parallel Colt), threads are used automatically when computations are done on a machine with multiple CPUs. Conversely, both Ooura's FFT and FFTW require manually setting up the maximum number of computational threads. Lastly, JTransforms' API is much simpler than Ooura's FFT, or even FFTW, since it is only necessary to specify the size of the input data; work arrays are allocated automatically and there is no planning phase.

3.7 Accuracy

There are two aspects about the accuracy of floating-point arithmetic in Java. The first is related to the internal design and implementation of Java's floating-point arithmetic. There are several flaws in this implementation [Kahan and Darcy 1998]. First of all, Java does not completely conform to the IEEE 754 standard, since it does not support the flags for IEEE 754 exceptions: Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result. In other words, no event occurs when the value of a floating-point number becomes either Infinity or NaN. Moreover, Java does not provide the IEEE extended precision, even though over 95% of today's computers have hardware that can support these types of numbers. Finally, of two traditional policies for mixed precision evaluation, Java chose the worse. However, our experience shows that Java's floating-point arithmetic is good enough for applications in image processing. This is supported by the fact that usually the pixels of an image are stored as integers (byte and short) or as a single-precision floats, thus the double (or even single) precision arithmetic provides a sufficient amount of accuracy.

Another aspect of the accuracy is related to the stability of an algorithm and round-off errors. In the previous release of Parallel Colt we observed inaccurate results for trigonometric transforms when the size of the input data was an integer with a large prime factor. The inaccurate results were caused by the mixed-radix FFT algorithm. When encountering a large prime factor, a slow, $\mathcal{O}(n^2)$, discrete Fourier transform algorithm was used. It is known [Schatzman 1996], however, that in these situations the root mean square error is $\mathcal{O}(\sqrt{n})$, where n is the size of the input data. The original FFTPACK library is also burdened with this error. In the current version of Parallel Colt (and JTransforms) we have fixed all the accuracy issues by implementing Bluestein's FFT

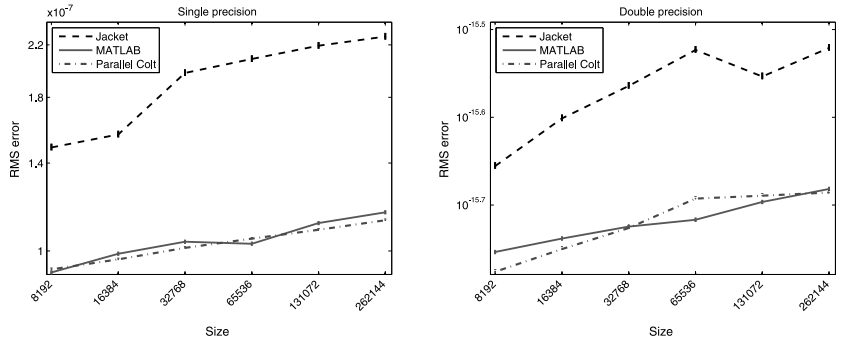

Fig. 1. Accuracy of complex, 1D FFT (power of two sizes). The vertical axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\sqrt{n}}$, where $\mathbf{x}$ is a vector whose size n is shown on the horizontal axis.

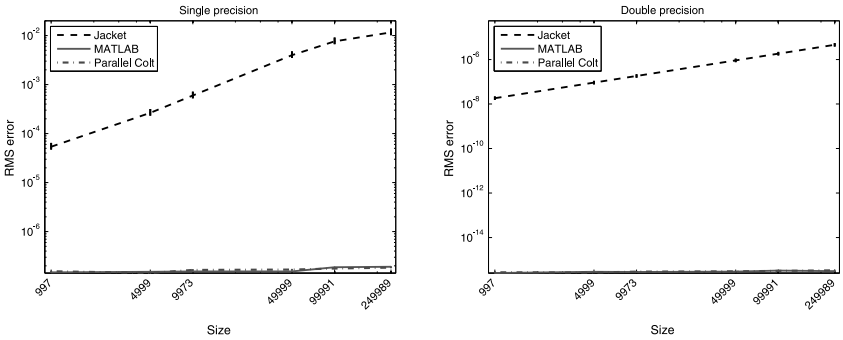

Fig. 2. Accuracy of complex, 1D FFT (prime sizes). The vertical axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\sqrt{n}}$, where $\mathbf{x}$ is a vector whose size n is shown on the horizontal axis.

algorithm [Bluestein 1968]. Figures 1 and 2 show that both single and double precision FFTs in Parallel Colt are as accurate as FFTs in MATLAB. Jacket's FFTs, on the other hand, are much less accurate when the size of the data is a prime number. Since the source code of CUFFT library (used by Jacket) is not available, we can only speculate that the Jacket's accuracy problems are also caused by the mixed-radix FFT algorithm. The length of the vertical error bars in these figures is equal to two standard deviation units.

3.8 Other Additions

To support new kinds of matrices described in Section 3.3 we had to implement some new data structures. In particular, we have added new hashmaps holding (key,value) associations of type (long –> double), (double –> long), (long –> float), (float –> long), (long –> long), (int –> long), and (long –> int). In addition, since the HyBR solver requires the 1-dimensional minimization routine (fmin), we have included the nonlinear optimization package [Verrill 2009] into our library.

Unit testing is an important part of every library, but it is absolutely crucial for mathematical software. Therefore, Parallel Colt contains a unit test

framework based on JUnit Community [2009]. The framework allows to write and run new test cases in a very intuitive way. We currently have available 6552 tests to check all the functionalities provided by sparse and dense matrices and iterative solvers.

Finally, a highly configurable benchmark framework for Parallel Colt has been developed. At this time we provide benchmarks for dense matrices (holding complex and real numbers) as well as for iterative solvers. In the configuration files, the user can define such properties as the number of threads, the size of the matrix, the number of repeats (to compute average time), and the path to the matrix file (stored in Matrix Market Exchange Formats). The timings computed by these benchmarks are automatically saved in text files.

3.9 Benchmarks

In the previous section we remarked that Parallel Colt is equipped with framework for performing benchmarks. Here we present benchmark results of two important computational kernels used in many scientific computing algorithms, and more specifically, in our target applications in image processing: FFT and sparse matrix-vector product.

3.9.1 *FFT*. FFTs are a key computational kernel in many image processing applications, including deconvolution, filtering, and image reconstruction. In this section we benchmark the performance of 2D real FFTs. As a testbed for our benchmarks we used a machine equipped with two Quad-Core Intel Xeon E5472 processors operating at 3.0 GHz, 32GB RAM, and an NVIDIA Tesla C1060. The system was running Ubuntu Linux 9.04 (64-bit), NVIDIA CUDA 2.3, MATLAB R2009b, AccelerEyes Jacket 1.2, Sun Java 1.6.0_16 (64-Bit Server VM) and ImageJ 1.43h. The following Java options were used: `-d64 -server -Xms20g -Xmx20g -XX:+UseParallelGC -XX:ParallelGCThreads=1`.

We benchmarked single and double precision, real input 2D FFTs in native MATLAB, MATLAB with Jacket, and Parallel Colt. The benchmarking methodology was adapted from FFTW [Frigo and Johnson 2009]. First, we run the *warm-up* phase (the first two calls require more time) which is not incorporated into the results. Then, we measured the FFT performance by performing repeated FFTs (100 times) of the same zero-initialized array. Benchmark results for Jacket include the time required for data transfer to and from the GPU memory. The maximum 8 threads were used in MATLAB and Parallel Colt. In addition, for Parallel Colt, one thread was used for the garbage collector, by specifying the flag `-XX:ParallelGCThreads=1`. It should be noted that the amount of GPU memory is a serious limitation for large-scale problems. On the hardware available for the tests reported here, the largest matrix size that fit into the GPU memory was 8192 × 8192 (4096 × 4096 for double precision). The benchmark results (in milliseconds) are reported in Tables II and III. For native MATLAB we report two results: the column MATLAB fft2() refers to the formula `B = fft2(A);` and the column MATLAB fft(fft()) corresponds the formula `B = fft(fft(A,[],1).',[],1).';`. Both of these formulas generate the same output (2-dimensional discrete Fourier transform), however, on our test machine, we have observed a significant performance difference between these

Table II. Performance in Milliseconds for Single Precision, Real Input 2D FFT

Size	MATLAB fft2()	MATLAB fft(fft())	Jacket	Parallel Colt
2000 × 2000	48.7	51.5	63.9	116.3
2048 × 2048	126.4	99.1	63.4	103.9
4000 × 4000	202.3	214.4	298.5	309.1
4096 × 4096	543.9	433.9	298.1	378.7
8000 × 8000	1877.3	887.7	1270.9	1599.9
8192 × 8192	2275.1	1785.9	1235.9	1707.5
16000 × 16000	10349.3	3851.2	-	10196.1
16384 × 16384	10140.3	7141.5	-	7502.7

Table III. Performance in Milliseconds for Double Precision, Real Input 2D FFT

Size	MATLAB fft2()	MATLAB fft(fft())	Jacket	Parallel Colt
2000 × 2000	79.5	90.3	156.5	136.5
2048 × 2048	206.2	154.6	134.0	122.4
4000 × 4000	439.4	405.2	691.0	466.5
4096 × 4096	878.7	633.5	537.9	483.7
8000 × 8000	3348.5	1699.4	-	3092.7
8192 × 8192	3858.8	2584.7	-	2412.9
16000 × 16000	15427.2	7421.1	-	14394.1
16384 × 16384	17088.5	10884.5	-	11488.1

two formulas. According to MathWorks' support this behavior occurs because of the *memory cache resonance* phenomena. The performance of all three libraries is comparable, with native MATLAB fft(fft()) having some advantages for larger data sizes. It should be emphasized that multithreaded FFTs were introduced in MATLAB R2009a; the performance of these routines in the previous versions was significantly lower. The other important conclusion that can be drawn from these results is that although the GPU-based single precision FFTs outperform both native MATLAB and Parallel Colt algorithms in most cases, they are much less accurate when the matrix dimensions are not a power-of-two numbers (see Figure 2).

3.9.2 *Sparse Matrix-Vector Product.* Sparse matrix-vector product is a key operation when iterative solvers are used to solve large-scale sparse linear systems. Here we present the performance comparison of this operation in MATLAB and Parallel Colt. MATLAB uses compressed-column format for storing sparse matrices. In Parallel Colt both compressed-column and compressed-row formats are available, but in this benchmark we only tested the former one. It should be emphasized that the latest release of MATLAB, R2009b, does not support single precision sparse matrices and its sparse matrix-vector product implementation is sequential. Parallel Colt, on the other hand, supports multithreaded sparse matrix-vector operations. However, there are two restrictions. When $\mathbf{A}$ is in compressed-column format, at most two threads are used to compute $\mathbf{y} = \mathbf{Ax}$, and when $\mathbf{A}$ is in compressed-row format, at most two threads are used to compute $\mathbf{y} = \mathbf{A}^T\mathbf{x}$. These limitations arise from the fact that it is not possible to split the job for these particular computations (with the associated storage format) into asynchronous tasks; all threads have to operate on

Table IV. Performance in Milliseconds for the Sparse Matrix-Vector Multiplications $\mathbf{y} = \mathbf{Ax}$ and $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ (numbers in brackets show the performance of $\mathbf{y} = \mathbf{A}^T\mathbf{x}$)

Matrix	Nonzeroes	MATLAB	Parallel Colt (double)	Parallel Colt (single)
Random	100,000	15.3 (11.3)	17.3 (6.9)	11.3 (4.3)
Rajat31	20,316,253	154.3 (140.1)	114.6 (61.9)	85.5 (36.8)
Nlpkkt120	95,117,792	545.4 (521.6)	266.6 (181.3)	153.8 (114.0)
S3dkq4m2	2,455,670	22.3 (21.5)	10.2 (8.0)	10.2 (5.5)

the whole vector $\mathbf{y}$. Therefore, in our implementation, if one of these situations occur and two threads are being used, then the first thread operates on the output vector $\mathbf{y}$ and the second thread works on its local copy of vector $\mathbf{y}$. A reducing addition is performed at the end of the multiplication. Our experiments have shown that using more than two threads for these cases slows down the performance.

The machine described in the previous section (with the same Java options) was also used for this benchmark. Table IV shows the performance of two operations $\mathbf{y} = \mathbf{Ax}$ and $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, where $\mathbf{A}$ is a sparse matrix and $\mathbf{x}$ is a dense vector. Four different matrices were used in this test. A random matrix (the first row in Table IV) was generated using the MATLAB command `A=sprand(1e6,1e6,1e-7)`, and the other three matrices come from the University of Florida Sparse Matrix Collection [Davis 2009]. It can be seen that Parallel Colt outperforms MATLAB for all tested matrices. This difference is much more significant for the transpose operation, where Parallel Colt can use more than two threads.

3.10 Application Programming Interface

Table V shows nine examples of different operations in MATLAB and in Parallel Colt. Since Java is a statically typed language, all variable names (along with their types) must be explicitly declared. MATLAB, on the other hand, is a dynamically typed language so there is no need to declare anything. An assignment statement binds a name to an object of any type and later the same name may be assigned to an object of a different type. This feature makes MATLAB expressions generally much more concise than the corresponding expressions in Java. Another essential difference between MATLAB and Parallel Colt arises from the inability to do operator overloading in Java (compare the matrix times vector expressions). Aside from these two differences, the expressions in Table V show that the same level of abstraction is used in MATLAB and Parallel Colt.

4. EXAMPLES OF USAGE

Although Parallel Colt is a general-purpose scientific computing package, we use it as a computational engine for the Java software tools we are developing for image processing applications. As a front-end to Parallel Colt we use the flexible and extensible Java imaging system ImageJ [Rasband 2009]. ImageJ is an open-source image processing program written in Java by Wayne Rasband at the U.S. National Institutes of Health (NIH). Besides having a large number of options for image processing applications, ImageJ is designed with

Table V. Comparison of MATLAB and Parallel Colt Expressions for a Sample Set of Matrix Operations

Description	MATLAB	Parallel Colt
New 2D dense matrix A	A = zeros(10, 10);	DoubleMatrix2D A = new DenseDoubleMatrix2D(10,10);
Copy of A	B = A;	DoubleMatrix2D B = A.copy();
Transpose of A	B = A';	DoubleMatrix2D B = A.viewDice();
Matrix times vector	B = A*x;	DoubleMatrix2D B = A.zMult(x);
2D FFT of A	B = fft2(A);	DComplexMatrix2D B = A.getFft2();
FFT along columns of A	B = fft(A,2);	DComplexMatrix2D B = A.getFftColumns();
Cosine of A (in-place)	A = cos(A);	A.assign(DoubleFunctions.cos);
Sum all entries of A	s = sum(A(:));	double s = A.zSum();
Location of max of A	[i, j] = find(A == max(A(:)));	double[] max = A.getMaxLocation();

a pluggable architecture that allows developing custom plugins; over 500 user-written plugins are currently available. Due to this unique feature, ImageJ has become a very popular application among a large and knowledgeable world-wide user community. Using ImageJ as a front-end to Parallel Colt allows for developing efficient and powerful plugins for image processing. Examples of two such plugins are described in this section: iterative deconvolution and motion correction of brain images obtained from Positron Emission Tomography (PET).

4.1 Parallel Iterative Deconvolution

In applications such as astronomy, medicine, physics, and biology, scientists use digital images to record and analyze results from experiments. Environmental effects and imperfections in the imaging system can cause the recorded images to be degraded by blurring and noise. Image restoration (sometimes known as deblurring or deconvolution) is the process of reconstructing or estimating the true image from the degraded one. Information about the blur is usually given in the form of a Point Spread Function (PSF). A PSF is an image that describes the response of an imaging system to a point object. A theoretical PSF can be obtained based on the optical properties of the imaging system. The main advantage of this approach is that the obtained PSF is noise-free. The experimental technique, on the other hand, relies on taking a picture of a point object, for example, in astronomy this can be a distant star.

Mathematically image deblurring is the process of computing an approximation of a vector $\mathbf{x}_{\text{true}}$ (which represents the true image scene) from the linear inverse problem

$$\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{true}} + \boldsymbol{\eta}\,. \tag{1}$$

Here, $\mathbf{A}$ is a large, usually ill-conditioned matrix defined by the PSF, $\boldsymbol{\eta}$ is a vector that models additive noise, and $\mathbf{b}$ is a vector representing the recorded image, which is degraded by blurring and noise. We assume that the PSF, and hence $\mathbf{A}$, is known, but the noise is unknown. Because $\mathbf{A}$ is usually severely ill-conditioned, some form of *regularization* needs to be incorporated [Hansen

1997; Vogel 2002] using, for example, a priori constraints. Examples include Tikhonov regularization and total variation.

Another approach is to use an (unconstrained) iterative method, and incorporate regularization through appropriate convergence criteria. Examples include CGLS (conjugate gradient for least squares) [Björck 1996], nonegatively constrained maximum likelihood methods, such as MRNSD (modified residual norm steepest descent) [Nagy and Strakoš 2000], and the Landweber algorithm (steepest descent with constant step size) [Bertero and Boccacci 1998]. It is also possible to combine variational regularization (e.g., Tikhonov) with an iterative solver. For example, HyBR (hybrid bidiagonalization regularization) [Chung et al. 2008] combines an iterative Golub-Kahan bidiagonalization method with singular value decomposition-based regularization. A distinctive feature of HyBR is that it can automatically choose regularization parameters and stop the iteration process based on the data. Note that because image deblurring problems are usually not symmetric, we use implementations applied (implicitly) to the normal equations, $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$, and thus the main cost per iteration is matrix-vector multiplications with $\mathbf{A}$ and $\mathbf{A}^T$. In this application the matrix is represented by the PSF, so $\mathbf{A}$ does not need to be constructed explicitly, and matrix-vector multiplications can be done efficiently using FFTs; see, for example, Nagy et al. [2004].

Parallel Iterative Deconvolution [Wendykier 2009f] is an ImageJ plugin that implements MRNSD, CGLS, HyBR, and a Wiener filter Preconditioned Landweber (WPL) algorithm. The first three methods are derived from RestoreTools [Nagy et al. 2004], and the Landweber algorithm is a parallel version of Iterative Deconvolve 3D [Dougherty 2005] with some enhancements. In addition, we have implemented two FFT-based preconditioners to accelerate convergence, which is generally necessary for MRNSD and Landweber. Parallel Colt is used as a computational engine for all implemented algorithms. In particular, in this application, we make use of the following features of Parallel Colt:

—multidimensional arrays to represent 2D and 3D images;
—trigonometric transforms for efficient matrix-vector multiplication and efficient preconditioner solves;
—linear algebra (e.g., SVD, QR least squares solver, Euclidean norm of a vector, various element-wise matrix operations).

The Parallel Iterative Deconvolution plugin has a clear and intuitive GUI (Figure 3), which includes eight drop-down lists (combo-boxes). The first two lists (*Blurred image* and *PSF*) provide an interface to input data. The next two lists (*Method* and *Preconditioner*) allow to select an algorithm used for deconvolution (*MRNSD*, *WPL*, *CGLS*, *HyBR*) and a preconditioner. Currently only the *FFT-preconditioner* is available (*WPL* uses a Wiener filter as a preconditioner). For stability, the preconditioner requires a spectral cutoff tolerance, which is computed automatically, via Generalized Cross Validation, by default (*Auto* check-box), but it is also possible to specify the value manually. In the *Boundary* combo-box the user can choose from three types of boundary conditions: *Reflexive*, *Periodic*, or *Zero*. The first type, reflexive, is usually the best

Fig. 3. GUI for Parallel Iterative Deconvolution.

choice. Other options include: threshold (the smallest nonnegative pixel value assigned to the restored image), resizing (size of padding: auto, minimal or the next power-of-two), different output types (same as source, byte, short, or float), show iterations, the number of threads, and batch processing (can be called from an ImageJ macro).

We remark that all this functionality is enclosed in a clear and intuitive GUI (Figure 3), where only options that are common for all methods are displayed in the main window (the advanced preferences are hidden under the *Options* button). In a typical usage scenario, there is no need to change the advanced preferences, since the default values are usually optimal. Moreover, by exploiting the computational efficiency of Parallel Colt, deblurring of large-scale 2D and 3D images can be obtained in seconds. For example, on the machine used in Section 3.9, a 2048×2048 2D image is deblurred within 9 seconds (5 iterations of HyBR), while a $128 \times 256 \times 256$ 3D image requires 26 seconds. Note that a $128 \times 256 \times 256$ 3D image corresponds to a matrix $\mathbf{A}$ that has dimensions $8{,}388{,}608 \times 8{,}388{,}608$, if it were constructed explicitly.

4.2 Motion Correction of PET Brain Images

Positron Emission Tomography (PET) is a technique used in nuclear medicine imaging for creating three-dimensional images of functional processes in the human body. Prior to a PET scan, radioactive materials are injected into a patient. Certain materials have known properties and may congregate near a certain physical structure in the area being scanned. This allows doctors and technicians an opportunity to check for certain abnormalities or diseases of the brain, heart, kidneys, or other organs. Here we focus on brain PET scans. Due to the radioactive nature of the injected materials, their nuclei are unstable. Positively charged particles, called positrons, are emitted. These positrons then travel a short distance before interacting with electrons. During an interaction, a positron and an electron are annihilated, with 511 keV photons emitted in opposite directions. If the patient is in a PET machine, which has

detectors that can detect the photons, these annihilations can be recorded. If two detectors detect photons within a certain amount of time of each other (called a coincidence event) those two photons are determined to have come from the same annihilation. These coincidence events may be recorded and kept in list form; an image of the brain may then be reconstructed from these events using projection techniques. If the patient does not move during the scan, the correct detectors will detect the annihilations. However, if the person's head has moved from its initial position, then the photons from an annihilation may be detected by detectors inconsistent with the expected position of the head. In these situations, it is important to use motion detection and correction schemes to improve the resolution of the reconstructed image. One approach currently being used in clinical settings involves attaching a target to the patient's head before the scanning begins. A motion tracker can then record the position and orientation of the head at many time intervals during the scan. One such system is the High-Resolution Research Tomograph (HRRT) PET scanner coupled with a Polaris Vicra optical tracking system [Faber et al. 2009].

There are different ways to use the motion information to improve the resolution of the reconstructed image. First, the coincidence events may be *rebinned* [Menke and Buckley 1996], which means that the motion information is used to assign each coincidence event to the proper detectors based on the location of the head at the time the annihilation occurred. In another method [Picard and Thompson 1997] multiple frames are acquired and each one is corrected for motion before creating a final image. Finally, the approach we consider here involves using the coincidence events as recorded to reconstruct an initial, motion blurred, image, and then solve an inverse problem of the form (1). However, in this case there is no PSF to define the matrix $\mathbf{A}$, but instead we must construct it explicitly, using a sparse data structure format.

To explain how the matrix is constructed, we assume the motion blurred image is the (normalized) sum of images at incremental times during acquisition. Each of the individual images represents a snapshot of the object in a fixed position. That is, if we assume the motion tracking device provides position information at m time instances, then the blurred image can be modeled as

$$\mathbf{b} = \sum_{\ell=1}^{m} w_\ell \mathbf{x}_\ell + \boldsymbol{\eta},$$

where $\mathbf{x}_\ell = \mathbf{A}_\ell \mathbf{x}$ is a vector representing the discrete image at time t_ℓ, w_ℓ is the normalization weight for the ℓth image (for example, we could simply use $w_\ell = \frac{1}{m}$), and $\boldsymbol{\eta}$ is additive noise. Furthermore, since we assume that the position of the object at time t_ℓ is known, we can construct the sparse interpolation matrix $\mathbf{A}_\ell$, which transforms the object $\mathbf{x}$ at a reference position to the new position represented by $\mathbf{x}_\ell$. Thus, the matrix modeling the motion blur is given by

$$\mathbf{A} = \sum_{\ell=1}^{m} w_\ell \mathbf{A}_\ell. \tag{2}$$

Note that the motion detection system used in our work provides the position information needed to construct the matrices $\mathbf{A}_\ell$. Since each $\mathbf{A}_\ell$ has a different

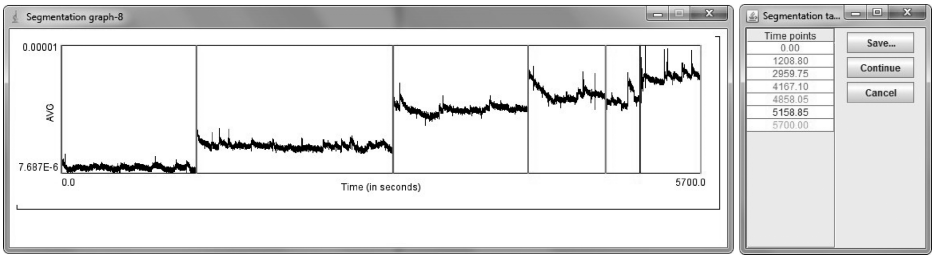

Fig. 4. Visual segmentation editor: A graphical tool for defining the segmentation of the motion information manually.

sparsity pattern, the overall sparseness of $\mathbf{A}$ decreases (that is, $\mathbf{A}$ becomes more dense) as more motion information is used. Thus, there is a significant trade-off between accurately modeling the motion blur and computational cost. To overcome this, we segment the position information into bins where the position of the object is essentially fixed, and compute an average position for each bin. Therefore, although the position tracking device may record, say, one thousand distinct head positions, in practice there may be only a few (e.g., 10) significantly different positions. Hence, the integer m in Eq. (2) denotes the number of bins, and the normalization weights ω_ℓ are determined from how the position information was segmented.

Parallel HRRT Deconvolution [Wendykier 2009e] is an ImageJ plugin for motion correction of PET brain images. Analogous to Parallel Iterative Deconvolution, this plugin implements four methods MRNSD, CGLS, and HyBR, as well as OSEM (Ordered Subsets Expectation Maximization) [Hudson and Larkin 1994], which is often used in medical imaging applications. However, instead of using FFTs, sparse matrix-vector products are the significant cost per iteration.

As previously mentioned, to optimize performance, it is important to segment the motion information into bins where the position of the head is essentially fixed. It is difficult to find an optimal, automatic segmentation of the motion information, so we have developed an easy way to adjust the segmentation. This integral part of the plugin constitutes the visual segmentation editor illustrated in Figure 4. This tool allows for editing an initial (automatically generated) segmentation in a very intuitive way. In addition, the main GUI for the plugin (see Figure 5) provides a choice between two interpolation schemes (Nearest Neighbor and Trilinear) to construct the matrices $\mathbf{A}_\ell$, and a *Solve* button that is used to solve the linear inverse problem. The *Solve* button allows the user to choose from the different algorithms and/or to choose the maximum number of iterations, and there are similar options to those described for the Parallel Iterative Deconvolution GUI.

This plugin utilizes the following features of Parallel Colt:

—multidimensional arrays to represent 3D images;
—dynamically resizing lists holding primitive data types for efficient implementation of interpolation algorithms;
—row-compressed sparse matrix format for storing matrices $\mathbf{A}_\ell$ and $\mathbf{A}$ as well as for efficient sparse matrix-vector multiplication;

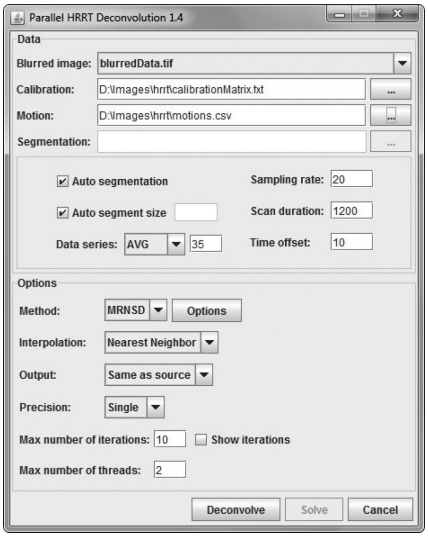

Fig. 5. GUI for Parallel HRRT Deconvolution.

—linear algebra (e.g., SVD, QR least squares solver, Euclidean norm of a vector, various element-wise matrix operations).

The performance of Parallel HRRT Deconvolution is comparable to the iterative deconvolution plugin. For example, on the machine used in Section 3.9, a $207 \times 256 \times 256$ 3D image is reconstructed within 48 seconds (5 iterations of HyBR, 13 segments). In this case, the sparse matrix $\mathbf{A}$ has dimensions $13,565,952 \times 13,565,952$, with 118,772,562 nonzeros.

5. SUMMARY

We have demonstrated the advantage of exploiting available hardware on modern, and affordable, computer architectures in scientific computing with multithreaded programming in Java. A significant contribution of our work is Parallel Colt, a multithreaded Java library for scientific computing and image processing. We have also described two ImageJ plugins for iterative image deblurring and motion correction of PET brain images as typical applications of this library. Thus, we are able to provide Java software to solve important problems in real image processing applications, and which can effectively make use of multicore CPUs available on affordable desktop and laptop computers.

Our results show that Java can be a competitive language for certain scientific computing applications. In addition, we observe that GPU-based FFTs currently perform worse than multicore CPU-based algorithms for vectors and matrices whose dimensions are not a power-of-two. Moreover, with Jacket, we were limited in the size of matrices that could be processed, and there are accuracy concerns associated with the FFTs used by CUDA. We also note that the sparse matrix-vector product operation in MATLAB is still sequential and slower than our Java implementation.

Although we expect the performance of MATLAB and Jacket to improve in future releases, we nonetheless believe that, with proper implementation, Java

software such as Parallel Colt will remain competitive for some scientific computing applications. Moreover, there is a need to develop high-quality scientific computing Java software. In addition to the image processing applications we have described in this articles, other open source projects that currently use Parallel Colt as a computational engine include the following.

—*NPAIRS/PLS-J* [Strother et al. 2009] is a Java program developed at Rotman Research Institute (Canada), capable of performing both nonparametric, prediction, activation, influence, reproducibility, resampling (NPAIRS) [Strother et al. 2002] and partial least squares (PLS) [Mcintosh et al. 1996] analysis.

—*TomoJ* [Messaoudi et al. 2007] is an ImageJ plugin for alignment, reconstruction, and combination of multiple tomographic volumes. It includes the most recent algorithms for volume reconstructions used in three-dimensional electron microscopy (the algebraic reconstruction technique and simultaneous iterative reconstruction technique) as well as the commonly used approach of weighted back-projection.

—*Incanter* [Liebke 2009] is a Clojure-based, statistical computing and graphics environment.

—*JQuantLib* [Gomez 2009] is a comprehensive framework for quantitative finance which provides a wide range of mathematical and statistical tools for the valuation of shares, options, futures, swaps, and other financial instruments. It also supports tools related to risk and money management.

—*Endrov* [Henriksson 2009] is a multipurpose image analysis program similar to ImageJ, but with some additional functionalities. Current features include 2D and 3D visualization of image data and annotation, visualization of 3D isosurfaces, support for large datasets (50GB+), data compression, infinite number of channels, batch processing, integration with external tools such as MATLAB image filters and analysis tools, nondestructive real-time application of image operations, and 5D regions of interest.

REFERENCES

ACCELERYES. 2009. Jacket. http://accelereyes.com/.

AHMED, N., NATARAJAN, T., AND RAO, K. R. 1974. Discrete cosine transfom. *Trans. Comput. C-23*, 1, 90–93.

AMERDO, B., BODNARTCHOUK, V., CAROMEL, D., DELBÉ, C., HUET, F., AND TABOADA, G. L. 2008. Current state of Java for HPC. Tech. rep. inria-00312039, INRIA.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., GREENBAUM, A., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, 3rd Ed. SIAM, Philadelphia, PA.

ARNDT, H., BUNDSCHUS, M., AND NÄGELE, A. 2009. Towards a next-generation matrix library for Java. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*.

BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND DER VORST, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* 2nd Ed. SIAM, Philadelphia, PA.

BERTERO, M. AND BOCCACCI, P. 1998. *Introduction to Inverse Problems in Imaging*. IOP Publishing Ltd., London.

BJÖRCK, Å. 1996. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, PA.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. 2002. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2, 135–151.

BLUESTEIN, L. I. 1968. A linear filtering approach to the computation of the discrete Fourier transform. *Northeast Electronics Research and Engineering Meeting Record* 10, 218–219.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. 2007. A class of parallel tiled linear algebra algorithms for multicore architectures. Tech. rep., Innovative Computing Laboratory.

BYOUS, J. 2003. Java technology: The early years. http://java.sun.com/features/1998/05/birthday.html.

CHUNG, J., NAGY, J. G., AND O'LEARY, D. P. 2008. A weighted GCV method for Lanczos hybrid regularization. *Elec. Trans. Numer. Anal. 28*, 149–167.

COOLEY, J. W. AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90, 297–301.

DAUTELLE, J.-M. 2007. JScience Project. http://jscience.org/.

DAVIS, T. 2009. The University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

DAVIS, T. A. 2006. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA.

DOOLIN, D. M., DONGARRA, J., AND SEYMOUR, K. 1999. JLAPACK - Compiling LAPACK Fortran to Java. *Sci. Program. 7*, 2, 111–138.

DOUGHERTY, R. 2005. Extensions of DAMAS and benefits and limitations of deconvolution in beamforming. In *Proceedings of the 11th AIAA/CEAS Aeroacoustics Conference*.

DRA SYSTEMS. 2000. OR-Objects. http://opsresearch.com/OR-Objects/.

FABER, T. L., RAGHUNATH, N., TUDORASCU, D., AND VOTAW, J. R. 2009. Motion correction of PET brain images through deconvolution: I. Theoretical development and analysis in software simulations. *Phys. Med. Biol. 54*, 3, 797–811.

FRIGO, M. AND JOHNSON, S. G. 2005. The design and implementation of FFTW3. *Proc. IEEE 93*, 2, 216–231.

FRIGO, M. AND JOHNSON, S. G. 2009. FFT benchmark methodology. http://www.fftw.org/speed/method.html.

GOLUB, G. H. AND LOAN, C. F. V. 1996. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press.

GOMEZ, R. 2009. JQuantLib project. http://www.jquantlib.org/index.php/Main_Page.

HANSEN, P. C. 1997. *Rank-Deficient and Discrete Ill-Posed Problems*. SIAM, Philadelphia, PA.

HARTLEY, R. 1942. A more symmetrical Fourier analysis applied to transmission problems. In *Proceedings of the Institute of Radio Engineer (IRE)*.

HEIMSUND, B.-O. 2007. Matrix toolkits for Java. http://ressim.berlios.de/.

HENRIKSSON, J. 2009. Endrov project. http://www.endrov.net/index.php/Main_Page.

HICKLIN, J., MOLER, C., WEBB, P., BOISVERT, R. F., MILLER, B., POZO, R., AND REMINGTON, K. 2005. JAMA: A Java matrix package. http://math.nist.gov/javanumerics/jama/.

HOSCHEK, W. 2000. Uniform, versatile and efficient dense and sparse multi-dimensional arrays. http://acs.lbl.gov/%7Ehoschek/publications/ACMJava2000.pdf.

HOSCHEK, W. 2004. Colt project. http://dsd.lbl.gov/%7Ehoschek/colt/index.html.

HUDSON, M. AND LARKIN, R. 1994. Accelerated image reconstruction using ordered subsets of projection data. *IEEE Trans. Med. Imag. 13*, 601–609.

JCUDA. 2009. JCuda project. http://www.jcuda.org/jcuda/JCuda.html.

JUNIT COMMUNITY. 2009. JUnit Project. http://www.junit.org/.

KAHAN, W. AND DARCY, J. D. 1998. How Java's floating-point hurts everyone everywhere. http://www.cs.berkeley.edu/%7Ewkahan/JAVAhurt.pdf.

LIEBKE, D. 2009. Incanter project. http://wiki.github.com/liebke/incanter.

MENKE, M. A. AND BUCKLEY, K. 1996. Compensation methods for head motion detected during PET imaging. *IEEE Trans. Nucl. Sci. 43*, 310–317.

MCINTOSH, A. R., BOOKSTEIN, F. L., HAXBY, J. V., AND GRADY, C. L. 1996. Spatial pattern analysis of functional brain images using partial least squares. *Neuroimage 3*, 143–157.

MESSAOUDI, C., BOUDIER, T., SORZANO, C., AND MARCO, S. 2007. TomoJ: Tomography software for three-dimensional reconstruction in transmission electron microscopy. *BMC Bioinf. 8*, 1, 288.

NAGY, J. G., PALMER, K., AND PERRONE, L. 2004. Iterative methods for image deblurring: A MATLAB object-oriented approach. *Numer. Algo. 36*, 1, 73–93.

NAGY, J. G. AND STRAKOŠ, Z. 2000. Enforcing nonnegativity in image reconstruction algorithms. In *Proceedings of the Mathematical Modeling, Estimation, and Imaging*, D.C. Wilson et al. Ed., Vol. 4121. 182–190.

NVIDIA CORPORATION. 2009. CUDA zone. http://www.nvidia.com/object/cuda_home.html.

OOURA, T. 2006. General purpose FFT (fast Fourier/cosine/sine transform) package. http://www.kurims.kyoto-u.ac.jp/%7Eooura/fft.html.

PICARD, Y. AND THOMPSON, C. J. 1997. Motion correction of PET images using multiple acquisition frames. *IEEE Trans. Med. Imag. 16*, 137–144.

RASBAND, W. S. 2009. ImageJ, U.S. National Institutes of Health, Bethesda, MD. http://rsb.info.nih.gov/ij/.

SAAD, Y. 1994. ILUT: A dual threshold incomplete ILU factorization. *Numer. Linear Algebr. Appl. 1*, 387–402.

SCHATZMAN, J. C. 1996. Accuracy of the discrete Fourier transform and the fast Fourier transform. *SIAM J. Sci. Comput. 17*, 5, 1150–1166.

STROTHER, S., MCINTOSH, A. R., SOMJI, I., ODER, A., SPRENG, N., WALLER, J., WONG, F., WRIGHT, D., YOURGANOV, G., AND ZHAO, R. 2009. PLS/NPAIRS-J project. http://code.google.com/p/plsnpairs/.

STROTHER, S. C., ANDERSON, J., AND HANSEN, L. K. 2002. The quantitative evaluation of functional neuroimaging experiments: The NPAIRS. *NeuroImage 15*, 4, 747–771.

SUN MICROSYSTEMS. 2004. New features and enhancements J2SE 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.

SWARZTRAUBER, P. N. 2004. FFTPACK5. http://www.cisl.ucar.edu/css/software/fftpack5/.

VANEK, P., MANDEL, J., AND BREZINA, M. 1996. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Comput. 56*, 179–196.

VERRILL, S. 2009. Nonlinear optimization Java package. http://www1.fpl.fs.fed.us/optimization.html.

VOGEL, C. R. 2002. *Computational Methods for Inverse Problems*. SIAM, Philadelphia, PA.

WENDYKIER, P. 2009a. CSparseJ project. http://sites.google.com/site/piotrwendykier/software/csparsej.

WENDYKIER, P. 2009b. JPlasma project. http://sites.google.com/site/piotrwendykier/software/jplasma.

WENDYKIER, P. 2009c. JTransforms project. http://sites.google.com/site/piotrwendykier/software/jtransforms.

WENDYKIER, P. 2009d. Parallel Colt project. http://sites.google.com/site/piotrwendykier/software/parallelcolt.

WENDYKIER, P. 2009e. Parallel HRRT Deconvolution project. http://sites.google.com/site/piotrwendykier/software/deconvolution/parallelhrrtdeconvolution.

WENDYKIER, P. 2009f. Parallel iterative deconvolution project. http://sites.google.com/site/piotrwendykier/software/deconvolution/paralleliterativedeconvolution.

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically tuned linear algebra software ATLAS. In *Proceedings of Supercomputing Conference*.

YIP, P. AND RAO, K. R. 1980. A fast computational algorithm for the discrete sine transform. *IEEE Trans. Comm. 28*, 2, 304 – 307.

ZHANG, B. 2005. Java FFTPack project. http://jfftpack.sourceforge.net/.

Received December 2008; revised January 2010; accepted April 2010